Efficient and Reusable Implementation of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers

分散記憶並列計算機のための効率的で再利用可能な細粒度マルチスレッディング及びゴミ集め

Kenjiro Taura

田浦健次朗

Efficient and Reusable Implementation of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers

分散記憶並列計算機のための効率的で再利用可能な
細粒度マルチスレッディング及びゴミ集め

Kenjiro Taura

April, 1997

Doctoral Dissertation

Department of Information Science
Graduate School of Science
University of Tokyo

Abstract

This thesis studies efficient runtime systems for parallelism management (*multithreading*) and memory management (*garbage collection*) on large-scale distributed-memory parallel computers. Both are fundamental primitives for implementing high-level parallel programming languages that support dynamic parallelism and dynamic data structures.

A distinguishing feature of the developed multithreading system is that it tolerates a large number of threads in a single CPU while allowing direct reuse of existing sequential C compilers. In fact, it is able to turn any *standard* C procedure call into an *asynchronous* one. Having such a runtime system, the compiler of a high-level parallel programming language can fork a new thread simply by a C procedure call to a corresponding C function. A thread can block its execution by calling a library procedure that saves the stack frame of the thread and unwinds stack frames. To resume a thread, StackThreads provides another runtime routine that rebuilds the saved stack frame on top of the current stack and restarts the computation from the blocking point. All these operations are implemented by using information already present on standard C stack frames, without requiring a frame format customized for a particular programming language. Experiments demonstrate that potential performance problems are not significant in practice, even on distributed memory computers in which each remote access causes a thread switch.

The developed garbage collection system is a simple mark & sweep collector that stops the user program while collecting. We show viability of such collectors on a large scale (up to 256 processors) distributed memory computer (Fujitsu AP1000+). Under a reasonable heap expansion policy, garbage collection occupies at most 15% of the application time (excluding idle time). More importantly, the overhead of garbage collection on parallel machines was, except for one application, in the ballpark of that on a single processor, indicating that garbage collection is at least as scalable as the applications. Another observation from the experiment is that independent local collection is a dangerous strategy which degrades performance

of synchronous applications severely (by up to 60%), contradicting previous believes that garbage collections should be done as independently as possible. This is because an independent local collection makes the collecting processor "unresponsive," making processors waiting for a reply from the collecting processor idle. For asynchronous applications with plenty of intra-node parallelism, independent collections perform better than synchronous collections, but the difference is small at least in our experiments. A more advanced strategy which adaptively selects a right strategy is also implemented and shown to be effective, though it is not significantly better than a simpler "always-synchronous" approach in the current experimental conditions.

On top of these runtime systems, a new programming language ABCL/f is designed and implemented. Several non-trivial applications written by the author and others are used for experiments. Both sequential performance and speedup of the applications are reported.

Acknowledgement

First of all, I express my biggest gratitude to my supervisor, Professor Akinori Yonezawa, for leading me to this very exciting area in computer science—programming language design, implementation, and parallel programming. Your encouragement was the greatest source of motivation to do my best, when I didn't know how to work as a researcher. It was the greatest event of me to become a research associate of your laboratory and enjoy extra time working in this amazing group.

Since I became a member of this group, Professor Satoshi Matsuoka has been the best advisor of my work. He has a broad range of background in this field and has been improving the quality of my job over five years.

One of the most fortunate things for a researcher is to be involved in a group with a lot of high ability persons. Dr. Masahiro Yasugi has been the greatest source of my inspiration ever since I begin to work. I learned from you how implementers should work and how to make systems faster. Professor Naoki Kobayashi, formerly a research associate of the Yonezawa Laboratory, has always been stimulating me with his keen intelligence. Luckily, after you take the current position, I still have joint meetings with your laboratory, to be timely informed of your activities. Since I begin to learn computer science, Hidehiko Masuhara has been the best friend to discuss with. You have been patiently listening to my incomplete questions or vague ideas and inspiring me with comments hitting the point.

As a research associate, there is nothing more exciting than working and discussing with motivated and hard-working students. I especially thank to Toshio Endo, Yoshihiro Oyama, and Takashi Ninomiya, for their good work and patience. You make my research life here far more fun and exciting than it would be without you. It is really difficult to motivate myself without working with other motivated people! I also thank Atsushi Igarashi,

Hirotaka Yamamoto, and Norifumi Gotoh for their good work and spontaneous devotion to the computing environment of the laboratory. Space prevents me of expressing thanks to everybody in the laboratory. I really thank all the people here who make this laboratory fun place to work.

I am grateful to Dr. Robert Halstead, for his comments and encouragement on the Chapter 2 of this thesis. Exchanging E-mails with you about this work was the most exciting events in my life as a researcher of this field.

I am thankful to the members of the thesis committee, Professor Takashi Chikayama, Professor Masami Hagiya, Professor Kei Hiraki, Professor Yoshio Oyanagi, and Dr. Mitsuhisa Sato, for their insight comments and criticism.

Not only research colleagues helped me. I wish to thank Keishi Tajima and Takashi Miyata, for all the enjoyable time spent with me.

At last, but not least, I wish to express my gratitude to Secretary Yoko Kobayashi for her support and all sort of arrangements. Without her help, it was impossible for me to submit this thesis by the deadline!

Contents

Chapter 1

Introduction

1.1 Presentation of Thesis

This thesis studies runtime systems for high-level programming languages on parallel computers. The primary focuses of the thesis are *multithreading* and *garbage collection*. We pursued these issues in the implementation of a concurrent object-oriented language ABCL/f on a distributed-memory parallel computer AP1000+ [34]. Techniques developed and observations drawn from the experiments are most relevant on large-scale distributed-memory multicomputers, though they are certainly useful in large-scale parallel computing in general.

Multithreading is a capability that manages a large number of threads of control in a single processor. With multithreading, the programmer can have much larger number of threads of control than the number of processors. The developed multithreading technique is unique in that it tolerates a very large number of (say, thousands) threads in each processor, while maintaining sequential speed and interoperability with existing C code. This is implemented in a small runtime system that can turn any standard C procedure call into *asynchronous* one. Having such a runtime system, the compiler's task becomes relatively straightforward; the compiler can use a C procedure call for a thread creation and just about any kind of C expressions for intra-thread sequential operations. In addition to the advantage that thread creation is fast, there is another advantage that it allows the compiler of high-level programming languages to generate *simple* C code that enables substantial optimizations performed by the C compiler. This

multithreading mechanism has been implemented on Sparc and Alpha.

Garbage collection is a capability that automatically reuses memory that is no longer used by the application. On distributed-memory multicomputers, data may be referenced from remote processors. Therefore, detecting if a region of memory is still used by the application requires substantial amount of cooperation between processors. Our garbage collector is a simple distributed mark & sweep collector, which judges if a datum is still live by reachability from the root. Despite its conceptual simplicity, implementations of this type of collectors on large-scale parallel computers are rare and viability of such collectors has not been well studied through experiments. This thesis proposes several implementation techniques for making such collectors feasible and demonstrates their viability through empirical study.

ABCL/f is a concurrent object-oriented language, designed and implemented on top of these substrates. It supports dynamic thread creation and concurrent objects, thus is suitable for applications that use dynamically created parallelism and data structures. Both multithreading and garbage collection are crucial underlying mechanism for implementing such programming languages.

1.2 Motivation and Background

As parallel computers become very widespread, problems people solve on such computers become very diverse. Experiments revealed that many problems have dynamic natures that make problem solving on large-scale parallel computers challenging. More specifically, many problems adopt *dynamic data structures* and extract *dynamic parallelism*.

Dynamic data structures generally mean data structures that are created dynamically (or incrementally) during the course of a computation. They are useful when the application needs a data structure whose shape, size, or distribution across processors is not known or difficult to approximate in advance, even if the primary parameters of the problem (*e.g.*, the problem size) are given. An unbalanced tree whose depth varies from one part to another and depends on the details of the input data is one such example. The most direct and flexible support for such applications is *dynamic object creation*, where the programmer can allocate a new block of memory at any

time in the computation. The allocated memory can be linked together to form a large structured data.

Dynamic parallelism generally means a parallelism that is dynamically created and extracted during the course of a computation. It is useful when the amount of parallelism that should be extracted to mask latency or to achieve a reasonable load balance is not known or difficult to approximate in advance, even if the primary parameters of the problem (*e.g.*, the problem size) are given. Parallel tree search problem with pruning is a typical example. In this problem, the amount of work under a given sub-tree cannot easily be estimated, even when the computation reaches the sub-tree. To achieve a reasonable load balance, one must divide the work into a much larger number of chunks than the number of processors and continues to distribute them across processors. The most direct and flexible support for such applications is *dynamic thread creation* (*or multithreading*), where the programmer can create a new thread of control at any time in a computation.

Efficient support for these dynamic applications imposes significant implementation challenges. To list some of important ones,

Efficient transparent data accesses: To support computation that uses dynamic data structures, the system should desirably provide *transparent* accesses to remote data, even if shared-memory is not supported by the hardware. Using dynamic data structures, inter-processor communication required by a computation is often unknown until runtime, because how data are distributed across processors are in general not known until runtime. This makes it hard to optimize communication cost by aggregating several remote accesses into a single message or by producer-initiated communication. If applications exhibit irregular data accesses, but relatively regular parallelism (*i.e., irregular data parallelism*), one can still apply runtime techniques such as inspector-executor [45]. Otherwise, one must resort to a straightforward implementation where a remote access implies a remote communication.

Automatic memory management: To support dynamic data structures, garbage objects (memory no longer used by the application) should desirably be automatically reclaimed by the system. Memory management by application programmers is already a source of trouble in single processor systems and thus many automatic management

systems are available. With dynamic parallelism, manual memory management becomes even more dangerous and error-prone, increasing the importance of automatic memory management. It is difficult, however, even for the sophisticated runtime systems to perform the job efficiently. In the presence of remote reference, detecting if a region of memory is still used by the application requires substantial amount of processor coordination.

Management of parallelism: To support dynamic parallelism, the system must tolerate much larger number of threads of control than the number of processors. This imposes a constraint on implementation that the resource requirement for a single thread must be small. In addition, the system should desirably support *fine grain* threads, not to constraint the way in which parallelism is extracted from the application. This means that the overhead of a thread creation as well as a thread switch should be small. Moreover, these facilities should be provided in a way that they do not hurt sequential performance.

Interoperability and Reusability: All these facilities should be provided in a way the resulting system can nicely inter-operate with existing software. Facing all the above challenging issues, it is tempting but not feasible to redesign and restructure the entire system from scratch. One must remember, however, that all the problems that appear in single-processor systems are still there and the total performance of the system can hardly be achieved without exploiting all these existing solutions. They include the whole optimization techniques implemented in sequential compilers, libraries written in sequential programming languages, and programming environment supports such as debuggers and profilers. We must avoid ending up with systems that implement some particular aspects of the above issues nicely, but fail to be utilized due to slow sequential speed or the lack of interoperability with exiting C libraries

1.3 Contributions

In this thesis, we address some of the above issues by building efficient runtime systems for multithreading and automatic memory management

for large-scale distributed-memory parallel machines, and by designing and implementing a programming language that provides easy and transparent accesses to these primitives. To list specific contributions of this thesis,

- It proposes a new implementation scheme for fine-grain software multithreading on stock microprocessors. The proposed scheme is unique in that it can make any stylized C procedure call an asynchronous one. Unlike traditional thread libraries, a thread creation needs neither an expensive startup procedure nor a large stack space. Unlike previously proposed efficient multithreading schemes, it does *not* assume a customized frame format designed for a particular programming language or a set of multithreading primitives. Instead, it operates on standard C stack frames and calling conventions. Difficulties arise due to calling conventions that assume sequential calls (*e.g.,* callee-save registers) and lack of information on C stack frames for multithreaded execution.

- It empirically studies performance of the multithreading scheme. The study shows that potential limiting factors of the proposed scheme do not become significant in practice.

- It shows an implementation scheme of distributed mark-and-sweep garbage collectors on large-scale parallel computers, together with several simple techniques that reduce the overhead of this type of collectors. It also demonstrates a design of the interface between a collector and an application that makes such collectors reusable across multiple language implementations. More specifically, the collector does not require extensive cooperation from the application program or the message-passing layer. This property should not be taken as granted, because mark & sweep garbage collectors must operate on a *consistent* global snapshot of the application, the definition of which includes messages that are in network or message buffer.

- It empirically studies performance of the proposed garbage collector through several benchmark applications. In a configuration that is the most space-intensive in our experiments, but is still not as intensive as some collectors used in heap-intensive programming languages,

garbage collections occupy at most 15% of the application time (excluding idle time due to load imbalance and communication) on 256 processors.

- It empirically demonstrates the importance of *scheduling strategies* of local collections. Contrary to previous beliefs that local collections should be independent, the experiments show that they should often be scheduled synchronously. Independent collection severely degrades performance of synchronous applications. Examination reveals that this is because independent collections introduce large scheduling skews in the applications. Since a local collection makes the collecting processor unresponsive to requests from other processors, processors that wait for a reply from a collecting processor also become idle unless they have parallelism which hide the latency introduced by the local collection. This can be avoided by performing local collections simultaneously on all the processors. We also developed an adaptive strategy that selects an appropriate local collection scheduling strategy by examining the behavior of the application.

- It shows design and implementation of a concurrent object-oriented language ABCL/f, as a running vehicle of the proposed implementation techniques. ABCL/f supports future and concurrent objects. Future is a means of dynamic thread creation and concurrent object is a means of location transparent data access and automatic mutual exclusion. By combining these primitives, the programmer can express data structures and parallelism needed by the application in a natural way.

- It shows how to implement a general and efficient communication through a first-class communication medium called channels. We also show that, on top of this mechanism, diverse calling sequences—any combination of local/remote and synchronous/asynchronous calls—are implemented efficiently and uniformly.

There are many issues that are important for supporting general purpose programming languages on large-scale parallel machines but are not tackled in this thesis. Amongst others, the multithreading mechanism explored in this thesis does not have provisions for migration on distributed memory

machines. In other words, once a thread is created, it executes on that processor. Migration is a desirable mechanism for supporting dynamic load balancing but is hard to implement, particularly on distributed-memory computers. Second, it does not address the first problem stated in the previous section—efficient location-transparent access to data. We assume that the cost model of distributed memory computers is directly exposed to the programmer or there is a shared-memory layer implemented by a lower-level software or hardware. In the implementation of ABCL/f, we take the former position. ABCL/f program automatically determines where a method is invoked based on the location of the receiver object but it does not perform any caching automatically. In other words, ABCL/f supports shared name-space, but does not support (any better approximation of) shared memory.

1.4 Evaluation Settings

We evaluated the performance of developed runtime systems as well as ABCL/f on single processor workstations and a distributed-memory parallel computer AP1000+ [34].

AP1000+ is a distributed-memory parallel computer. The processor elements are SuperSparc 50 MHz with 16 MB physical memory. The system we used in our experiments has 256 processors. Processors are connected via a torus network whose peer-to-peer bandwidth is 25 MB/s.[1] We used a bundled send/receive communication library. The minimum latency + overhead of a round-trip communication between a pair of processors with the library is about 40μs, or 2,000 processor cycles. The default operating system for the AP1000+ is a single-task operating system with no virtual memory support. A parallel job monopolizes all the CPUs and memories. More importantly, a message is never delayed by scheduling skew between the sender and the receiver.

For evaluations shown in Chapter 2, 3, and 6, we use application programs listed in Table 1.1 or a part of them. Characteristics of these applications are described as necessary in each chapter. A through description for BH, CKY, and RNA are given in Appendix A.

[1] AP1000+ is equipped with two additional networks for broadcast and communication with a host (frontend) processor. They were not used in our experiments.

Application	Description
BH	Nbody simulation by Barnes-Hut Method
CKY	Parser for Context Free Grammars
RNA	RNA secondary structure prediction by tree search
GA	Genetic algorithm

Table 1.1: List of parallel applications

BH is a parallel N-body simulation using a hierarchical tree structure. The original sequential algorithm is published in [9] and there are many parallel formulations such as [32, 75, 76], most of which are written in C. CKY is a parallel parsing algorithm for context-free grammars. The original sequential CKY algorithm was proposed by Cocke, Kasami, and Younger [43]. A survey of parallel algorithms is given in [50]. Refer to [54] for our algorithm. RNA is a parallel tree search program that predicts the secondary structure of an RNA molecule from a given sequence of bases. The original program was written by Nakaya in C using message passing [49] as well as a concurrent object-oriented language Schematic [72]. A simplified version is written in ABCL/f by the author and used in the evaluation in this thesis. GA is a parallel genetic algorithm written by Hiyane [35].

1.5 Roadmap

The rest of the thesis is organized as follows. Chapter 2 describes the developed multithreading techniques in a language-independent fashion. Details that are relevant only for ABCL/f are left for Chapter 5. Chapter 3 devotes to garbage collection. Chapter 4 focuses on the design of ABCL/f. Chapter 5 then presents implementation of ABCL/f. Since the important aspects of the implementation of ABCL/f have already been presented in Chapter 2 and 3, this chapter mainly focuses on mappings from particular language constructs supported by ABCL/f to facilities provided by these runtime systems. Chapter 6 demonstrates applications written in ABCL/f and examines its performance. Finally, we summarize the work and state conclusions in Chapter 7.

Chapter 2

Multithreading

Compiling into C is increasingly becoming an attractive approach to implementing high-level programming languages, for its portability and potential performance advantage thanks to optimizations performed by C compilers. However, it is difficult to map the execution model of multithreading languages (languages which support fine-grain dynamic thread creation) onto the single stack execution model of C. Consequently, previous work on efficient multithreading uses elaborate frame formats and allocation strategy, with compilers customized for them. This chapter seeks an alternative cost-effective implementation strategy for multithreading languages that can maximally exploit current sequential C compilers. We identify a set of primitives whereby efficient dynamic thread creation and switch can be achieved and clarify implementation issues and solutions which work under the stack frame layout and calling conventions of current C compilers. The primitives are implemented as a C library and named StackThreads. In StackThreads, a thread creation is done just by a C procedure call, maximizing thread creation performance. When a procedure suspends an execution, the context of the procedure, which is roughly a stack fame of the procedure, is saved into heap. Contexts saved into heap are reconstructed on top of the C stack when the thread is resumed. With StackThreads, the compiler writer can straightforwardly translate sequential constructs of the source language into corresponding C statements or expressions, while using StackThreads primitives as a *blackbox* mechanism that switches execution between C procedures.

2.1 Introduction

Many parallel programming languages support dynamic creation of threads. Example language constructs include futures [33], asynchronous message passing between concurrent objects [2, 79], fork-join [51], parallel blocks and loops [17, 20], and implicit parallelism [52]. Implementation of parallel languages with dynamic thread creation (which we hereafter refer to as *multithreading languages*) must achieve efficient multithreading without sacrificing good sequential performance. Compiling multithreading languages into C and exploiting optimizations performed by the C compiler is an attractive choice for obtaining sequential performance. It has been challenging, however, because the execution models of multithreading languages are not naturally mapped onto the execution model of C, which assumes a single stack. The time and space overhead of allocating a separate stack for each thread is prohibitively large, hence existing user-level thread libraries [23, 44] cannot straightforwardly be used for implementing multithreading languages. Consequently, most of the previously published efficient implementations of multithreading languages adopt a custom frame management and generate either assembler or assembly-like C code in which frame management and context switch code sequences are inlined. In such implementations, a thread creation typically allocates only a single frame from a general free storage and a context switch just saves live registers on the frame and transfers control to the restarting thread [24]. While this approach achieves very fast thread creation and context switch, there are several disadvantages and potential pitfalls. First, the compiler development cost is very high, because they must design runtime data structures from scratch for threaded execution and the compiler must perform low-level analyses such as live-range analysis to emit inlined context switch sequences. Second, sequential performance is sacrificed unless there is substantial effort on optimization. They must implement many sequential optimizations when generating assembly. Even when generating C code, C compilers often fail to optimize assembly-like C code because of its highly complex and unstructured representation of computation. For example, restarting a computation inside a loop requires a goto statement into the body of the loop, which is likely to disable optimization by the backend.

This chapter presents an attractive alternative for efficient implementation of multithreading languages. The mechanism is provided as a *low level C library*, called StackThreads. By *low level*, we mean that only primitive frame management mechanisms are defined by the library. Supporting higher-level abstractions on top of the base primitives is left for language designers and implementers. Section 2.6 demonstrates several example abstractions built on top of StackThreads. By *library*, we mean that most of the work needed for multithreading is done *under cover* of the library, without requiring extensive cooperation from the code generator. More precisely, the generated code simply calls a few library routines when a thread blocks. The library routine performs all the work needed to switch to another thread. In particular, the context-saving sequence does not have to be inlined in the generated C code. Hence, with StackThreads, sequential constructs can be straightforwardly compiled into corresponding C statements or expressions which may call a library routine when evaluation can no longer continue. This reduces development cost and enhances the chances for the backend optimization.

Unlike traditional thread libraries, StackThreads meets the performance requirement of multithreading languages—small creation overhead. In StackThreads, starting a new thread, including parameter passing, takes only a few instructions. In fact, starting a new thread that executes the body of a C function f is just a procedure call to f (possibly with extra parameters, depending on the implemented language construct). In the case where a thread blocks, mechanisms are provided to (1) save the context of the C procedure and resume the caller of a procedure, and (2) later restart a blocked thread from a saved context. Unlike previous implementations of multithreading languages in which these or similar mechanisms are implemented on top of a customized frame management protocol, StackThreads implements the mechanisms which work with any C code satisfying the few conditions described in Section 2.4.5. That is, the generated C code uses conventional C stack frames and procedure linkage conventions including parameter passing, result value passing, and even callee-save registers. Our primary contribution is to identify a set of primitives which are *required* for efficient multithreading and *implementable* under the stack frame format and calling conventions of current sequential compilers. The rest of this chapter is organized as follows. Section 2.2 reviews previous work on software implementations of multithreading. Section 2.3 summarizes the

stack frame layout and code generation conventions of C procedures which StackThreads mechanism relies on. Section 2.4 outlines how our mechanism works and Section 2.5 describes implementation details and machine-specific issues. Section 2.6 demonstrates several higher-level constructs built on top of StackThreads. Section 2.7 reports performance numbers and Section 2.8 summarizes this chapter.

2.2 Related Work

This section reviews previous efficient multithreading schemes. We limit our focus to schemes that are implemented on conventional CPUs; We do not discuss those that are implemented on multithreaded architectures. All of them either involve custom frame management and procedure linkage conventions or restrict the concurrency model so that they can be implemented without a general multithreading mechanism. Table 2.1 lists these works in roughly chronological order with their supported concurrency models and code generation schemes. A concurrency model is *general* if they implement a general multithreading model and *restricted* otherwise. By *general* multithreading model, we mean that the system guarantees that created threads are scheduled eventually, at least when it becomes the only runnable thread.[1] A code generation scheme is *native* if it generates assembly, *assembly like C* if it generates C with inlined frame management and context switch code sequences, and *simple C* if it can simply run on top of C's stack frame management.

Most notably, schemes that adopt simple C code generation—leapfrogging [74] and lazy RPC [28]—do not support a fully general concurrency model. A distinguishing feature of StackThreads is that it allows simple C code generation while implementing a general multithreading model. Below we classify previous work into three categories and describe each work in more detail. The three categories include those that use a simple task pool for thread management, those that adopt more elaborate and complicated thread management schemes, and those that simply run on top of C's stack frame management at the cost of a restricted concurrency model.

[1] Most work, including ours, do not guarantee any stronger sense of fairness. They only guarantee that neither the runtime nor the compiler add dependencies between threads which are otherwise independent.

	Concurrency Model	Code Generation Scheme	Primary Target Language
Threads in SML/NJ [22]	general	native	SML/NJ [4]
LTC [29, 48]	general	native	Multilisp [33]
TAM [24]	general	native	Id [52]
ABCL/AP1000 [70]	general	assembly-like C	ABCL
Leapfrogging [74]	restricted	simple C	Multilisp
Olden [59]	restricted	native	Olden
Concert [57]	general	assembly-like C	ICC++ [20] and CA [19]
Lazy Threads [31]	general	native	Id
Lazy RPC [28]	restricted	simple C	ParSubC [28]

Table 2.1: List of previous efficient implementations of multithreading

2.2.1 Thread Management by Simple Task Pool

Many multithreading implementations use a custom procedure linkage convention and frame management strategy, and generate assembly or assembly-like C code in which frame management and context switch code are inlined. The simplest management scheme allocates activation frames from a general free storage (e.g., free list) on an invocation-by-invocation basis, so that each thread does not have to have a stack. All runnable threads are stored into a task pool. At thread creation only allocates a thread control block, which does not have to have a stack, and stores it into the task pool. When a thread blocks, it simply schedules another thread from the task pool. TAM [24] and threads implemented in SML/NJ [22] fall into this category. TAM allocates an activation frame for each parallel invocation of a function or a loop body, from a free list. SML/NJ [22] implements threads using the callcc primitive. Since SML/NJ allocates all activation frames from the heap [5, 66], callcc is implemented simply by capturing the pointer to the current frame and saving callee-save registers. Hence, SML/NJ's thread management using callcc effectively implements simple heap-based frame management very efficiently.

This strategy necessarily uses a custom frame format as well as calling convention, and calling legacy libraries written in C needs special setup

procedures such as a stack switch. In addition to such software engineering issues, threads in this category have some performance disadvantages. First, thread creation incurs expensive operations such as allocating frames from a general free storage and enqueueing a frame into a task pool. Second, they usually have no chances to pass the result value of an invocation via a register. The result value is always written into memory, because the callee in general does not know when the caller is scheduled. Since registers are volatile storage, returning the result value via a register requires making some assumptions on the scheduling order and exploiting them. Third, they have no provisions for using callee-save registers on a thread creation. The caller saves all its contexts prior to a thread creation. This is again because no scheduling order is assumed between a parent thread and a child thread.

2.2.2 More Elaborate Thread Management Schemes

More elaborate thread management schemes are based on the observation first stated by Mohr, Kranz, and Halstead [47, 48]. The observation is that a thread creation merely has to leave minimal information to perform a real fork retroactively when it turns out to be really needed. More precisely, when we create a thread that evaluates the body of f, we continue the evaluation of f just as a sequential call.[2] Mechanisms are provided to resume the continuation of f without waiting for its completion in case it is blocked. If f is not blocked at all, the cost of a thread creation is roughly that of a procedure call + writing a descriptor to indicate a potential fork point. This basic structure—minimal fork overhead and retroactive work generation—can be ubiquitously found in many works which follow [31, 57, 59], in different contexts with further clarifications and improvements. This basic idea opens the door to several improvements over the simple task-pool approaches to managing threads. First, allocating a frame of a new thread from a stack is more efficient than allocating from a general free storage and enqueueing it to a task-pool. Moreover, this process is very similar to a procedure call in sequential languages, giving us an opportunity to express a thread fork in C's procedure call. Second, since the scheduling order is fixed and LIFO, they can return the result value via a register when a thread terminates

[2] A thread creation in LTC leaves a description a task pool so that another processor can steal it. This is a cost of dynamic load distribution and not a cost of multithreading *per se*.

without blocking, because we know the next thread to run will be the caller. Protocols must be devised for returning the result value after a thread is blocked. Third, we have an opportunity to exploit callee-save registers even on a thread creation. This is again because, when the callee terminates without blocking, we know the caller is scheduled immediately after the callee, enabling the callee to restore the callee-save registers for the caller. The execution scheme of the present work is based on the same observation and most close to that of the authors' previous work [70] and the hybrid execution model of Concert [57]. The difference between our work and any prior work is *where* the mechanism is implemented. Existing schemes use a custom frame management protocol and do not allow the direct reuse of a sequential compiler substrate—optimizing C compilers. On the other hand, StackThreads deals with a conventional C stack. Compilers of higher-level languages can straightforwardly map sequential computation onto C and a context switch simply calls library routines at runtime.

Differences between schemes in this category lie in how to deal with *blocking*—situation where the current thread can no longer proceed. When a thread that evaluates a procedure f blocks, [57] and [70] simply save the stack frame of the procedure and resume the frame just below the current frame. Both works implement this mechanism by generating assembly-like C code.

A procedure in the source language is, ignoring inline expansion, compiled into a C procedure. For each potential blocking point, the compiler generates a code sequence that saves all live variables into a heap frame and returns. Another code sequences is generated for restarting a computation from a blocked point. It loads live variables from memory and "goto" the blocked point. At least in the authors' experiences in [71], this implementation strategy is not so successful in terms of cost-effectiveness. First, the generated C code is very large and has a very obscure control/data flow due to inlined switch sequences. Hence, C compilers are likely to fail to optimize them. Second, compiler development cost is high, because we must perform low-level analysis such as live-range analysis to guarantee correct execution. Both factors lose some of the motivations for generating C code¿ease of compiler development and backend optimization.

LazyThreads [31] takes a similar but different approach to frame management and thread suspension. Frames are allocated in the unit of what

they call *stacklet*. A stacklet is a contiguous region that can hold several frames, but is much smaller than the typical stack size. A blocked thread resumes a caller without copying the stack frame to heap. Instead, each procedure checks if space is available on top of the current frame. If not, anew stacklet is allocated, regardless whether the call is sequential or parallel. Implementation of LazyThreads necessarily has to design runtime data structure from scratch; the implementation was done by modifying the GNU C compiler.

2.2.3 Simple C Code Generation with Restricted Concurrency Model

Some multithreading languages implement multithreading on top of the stack frame management mechanism of C. The basic idea is we continue execution of a single thread as far as we can and, when a thread is blocked, we *grow* the stack by other schedulable work, rather than unwinding the stack. In this way we can hold contexts of multiple threads in a single stack without stack unwinding, which cannot be naturally expressed in C.

Leapfrogging [74] implements Multilisp's future construct. A thread which evaluates an expression e is created by a future expression (future e). When a processor encounters a future expression, the processor continues to evaluate the *continuation* of the expression, leaving a descriptor of e in a shared task pool.[3] A task is blocked when it needs the value of an undetermined future. When a processor executing a task T encounters an undetermined future f, the processor now steals work from the shared task pool, but only steals one which is a subtask of f (including f itself). The stolen subtask (call it F) *on top of the current stack*. This strategy can be naturally expressed in C's stack frame management mechanism. The processor simply fetches F and calls a procedure that evaluates F'. An implication is that the blocked thread T can only be resumed after the evaluation of F finishes. This scheduling is not always safe. It is safe as long as determining the value of F requires determining the value of F', because in this case resuming T, which we know requires the value of F, transitively

[3]Using a shared task pool does not imply leapfrogging assumes shared memory. The shared task pool can actually be implemented as a logically shared, but physically distributed data structure.

requires determining the value of F. To summarize, leapfrogging can evaluate two tasks in a single stack as long as the lower thread has been blocked and is dependent on the upper thread (assuming a stack grows upwards). Evaluating multiple tasks that may be independent requires correspondingly many stacks. The concurrency model of leapfrogging is more restrictive than the general multithreading model in several ways. First, it does not support speculative computations. As the authors pointed out in [74], if F' is speculative and does not contribute to the value of f, leapfrogging causes a deadlock which should not occur in a general multithreading model. Second, it only supports 1-producer-N-consumers synchronization via future and does not support general synchronization primitives. More specifically, it assumes that a blocked thread knows the resumer thread (the thread that is going to resume it). This condition holds in stylized uses of futures, but does not hold for general synchronization primitives such as mutexes and condition variables. Finally, it does not encourage eager movement of tasks or application-specific task placement, which is particularly importation on distributed-memory parallel machines. This is again because independent tasks may not coexist on a single stack. A new task can be evaluated only when the current task is blocked or an empty stack becomes available. Lazy RPC [28] is based on the same idea. The difference is that, when a task blocks, the processor steals *any* task in the task pool. The discussion and restriction above also apply for Lazy RPC. StackThreads, on the other hand, implements general multithreading with a single stack (or a constant number of stacks) per processor. The key mechanism is stack unwinding, in which we can speculatively evaluate independent tasks in a single stack and revoke speculative decisions when the topmost task is blocked. Unlike leapfrogging or Lazy RPC, StackThreads moves stack frames, thus is incompatible with C programs or C compiler optimizations which assume they do not move. The primary use of StackThreads is therefore as a compiler target, rather than for user-level libraries for C programs.

2.3 Procedure Linkage Convention of C Procedures

2.3.1 Stack Frame Layout

This section summarizes procedure linkage and code generation conventions of C procedures, which are necessary for understanding the details of StackThreads. In particular, understanding the details of the procedure return mechanism—how a procedure returns to the caller so that the caller continues execution—gives us opportunities for saving/restoring the context of procedures in an unusual way. The background includes stack frame layout, register usage convention, how the linkage between a caller and a callee is maintained, and when/where registers are saved.

Figure 2.1 shows a typical stack frame layout of a C procedure. The figure illustrates a stack frame for a procedure f and its parent P, assuming the stack grows downwards. The lowest and the highest addresses of the stack frame are pointed to by the stack pointer (SP) and frame pointer (FP), respectively. A stack frame for f holds:

- local and temporary variables of f,
- callee-save registers for P, and
- the link to P, which is the return address and the frame pointer of P.

Incoming parameters not allocated to registers are stored in the caller's frame, so that the caller does not have to know the frame layout of the called procedure. The offset of the incoming parameters from the callee's frame pointer is constant across all procedures, so that the callee does not have to know the caller. It is the callee's responsibility to restore the SP and FP of the caller upon procedure return.

2.3.2 Register Usage Convention

The register usage convention for a processor classifies CPU registers into two categories. One is callee-save registers, which the caller assumes are preserved across a procedure call, and the other is caller-save registers, which

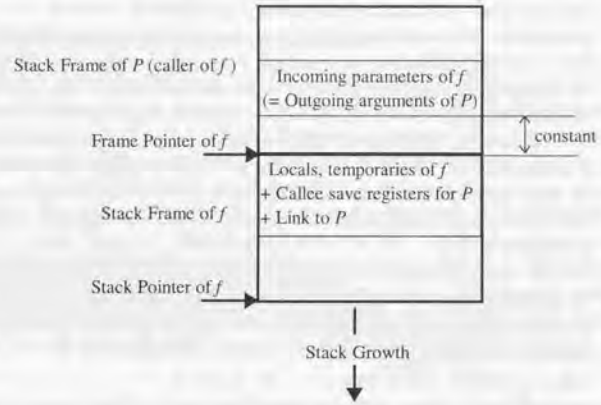


Figure 2.1: A typical stack frame layout of a C procedure. Parameters are in the frame of the caller. Local variables, temporary variables, callee-save registers, and links to the caller (*i.e.*, return address and FP of the caller) are in the frame of the current procedure.

the caller assumes are destroyed across a procedure call.[4] In order for a procedure to return to the correct place with the correct values in registers, a stack frame saves the return address, the frame pointer of the parent, and the callee-save registers which it destroys. When a procedure returns, it restores the values of the frame pointer, stack pointer, and the callee-save registers and jumps to the return address. The caller then continues execution, *assuming FP SP and callee save registers have the original values and other registers do not*. In other words, FP, SP, and callee-save registers constitute f's *context*—information that must be restored when f is rescheduled. The basic idea behind any thread library is that whenever we fork or switch a thread, we save enough information so that we can restore the context from the point where we reschedule the thread. Since the point where we reschedule the thread is usually unknown, ordinary thread libraries save *all* contexts, including *all* callee-save registers in the calling convention, of the current thread whenever a fork or a switch occurs. StackThreads, on the other hand, does not save all contexts on a thread fork; when a procedure f is forked, it saves exactly the same amount of context as an ordinary procedure call to f.

This makes a thread fork efficient but raises a difficult question against the implementation of StackThreads, because the callee-save registers for a procedure may be spread into an unknown number of frames and registers. Suppose a procedure f is using four callee-save registers A, B, C, and D, calls a procedure g that uses A and B, which in turn calls a procedure h that uses B, C. Where is the relevant context for f? When h is active (*i.e.*, its frame is on the top of the stack), A and B are saved in g's stack frame, C in h's stack frame, and finally, D still in the register! To save f's context and resume it later, we must find where they are. Information is not present in stack frames as to which callee-save registers are used by a procedure or where they are saved. Even if it were present, interpreting information and restoring registers would make context switch prohibitively slow. The way in which we handle callee-save registers instead relies on an assumption about the code generation style of C compilers. The assumed code generation style of C compilers is that a procedure saves callee-save registers at the entry (or

[4] SP and FP are also assumed to be preserved across a procedure call. For our purpose, however, we consider them as special registers and distinguish them from regular callee-save registers.

prologue) of the procedure and restores them at the exit (epilogue) of the procedure, *all at once*. In other words, a procedure does not incrementally save them depending on the control path. The assumption is true of all the compilers we know of including GNU C compilers. The programmer's manual of Mips [42] explicitly states that callee-save registers are saved at entry. This assumption validates an interesting technique for blocking a thread and resuming the parent in a consistent state, called *epilogue code threading*, which is further described in Section 2.4.3.

2.3.3 Summary: Where is Context?

When a procedure invocation I is inactive (*i.e.*, its stack frame is not at the top of the stack), the relevant context for I consists of: (1) the local and temporary variables of I, (2) the incoming parameters of I, (3) the stack pointer and frame pointer of I's frame, (4) and the callee-save registers of I. Locals and temporaries are in I's frame; incoming parameters are in the frame of I's caller. The frame pointer is saved in the frame of the direct child of I (unless the direct child does not save it at all). The stack pointer is the frame pointer of the direct child of I. Hence it may still be in the register or saved in the frame of the grandchild of the thread. Finally, callee-save registers are hard to locate. The next section further details how to locate the first three constituents and how to capture the callee-save registers.

2.4 StackThreads: Framework and Implementation

2.4.1 Overview

The basic execution scheme of StackThreads is simple and has already been published elsewhere by the authors [71]. When we fork a new thread that evaluates a procedure f, we call f just as a sequential call. When f blocks, it can resume its caller by moving its frame from the stack to the heap and unwinding the stack. Since the caller can be rescheduled even if the callee blocks, we effectively create a new thread of control in this sequence. When f is later rescheduled, the context is restored on top of the stack and control transfers to the point where f blocked. What needs to be clarified is

which part of a stack frame and registers must be saved/restored and how to correctly capture them from conventional C stack frames.

2.4.2 Creating Threads by C Procedure Calls

Suppose we wish to fork a new thread which evaluates the body of a C procedure f. The parent of the thread just calls f, passing parameters in exactly the same way as normal C procedure calls. If f successfully terminates its execution without blocking, the result value is obtained as the return value of the procedure call. In StackThreads, however, control may return to the caller even if f does block, in which case the return value from f is unspecified. Once f has blocked, it does not make sense for f to return the result value by means of C's return statement, as the caller may no longer be scheduled immediately after the return. Hence, it is often necessary for a thread to tell the caller whether it terminated or blocked, and, if blocked, the location through which these two threads thereafter communicate. In particular, so-called sequential calls must be implemented using this kind of protocol, if the calls may be blocked. StackThreads does not define any fixed protocol for this, based on the observation that an appropriate protocol is often language dependent and sometimes unnecessary. The protocol is, for example, unnecessary in pure Actor-based languages where all method invocations are done via an asynchronous message and the result value is passed via another asynchronous message. Section 2.6.3 shows example protocols for passing the result value for a future-like communication primitive.

2.4.3 Blocking a Thread by Epilogue Code Threading

StackThreads provides a way in which a thread saves its context into heap and resumes the frame just below the current frame (the *current parent* of the thread). The current parent is the original caller (creator) when the thread blocks for the first time. When a thread A was blocked and another thread B later resumes A, B becomes the current parent for A. Notice that StackThreads by no means forces the runtime system of the language to resume its current parent immediately when a thread blocks. It may spin a while, try to find other work locally or from the network, or even run the garbage collector when appropriate, and thread libraries alone cannot determine the right action. This is the reason why StackThreads supports

the parent resuming as a library, rather than as a built-in response to a thread blocking.

Suppose a thread P forked f, which now wishes to resume P again. It allocates a heap frame of appropriate size by calling library function `alloc_ctxt` and then calls `switch_to_parent` to trigger the actual context switch, passing the allocated frame to `switch_to_parent`. The procedure `switch_to_parent` fills the heap frame with the context of f and resumes the current parent of f. When f is resumed later, control returns to f as if `switch_to_parent` returned normally. The allocation and the actual context switch are separated because the context switch code needs to perform a language-specific action on the context (e.g., storing the pointer to the context somewhere for later resumption). The typical context switch sequence is to first allocate a heap frame, save the pointer to the frame somewhere to implement the language construct, and then call `switch_to_parent`. This interface also allows the language implementers to reuse the same memory for saving context over multiple suspensions in a single procedure.

Let us see how the procedure `switch_to_parent` works. For now, let us make an assumption for simplicity, which we will relax later, that f *directly* calls `switch_to_parent`, thus `switch_to_parent` knows the frame of the blocking thread is just below the current frame. Stack frames and the control flow when `switch_to_parent` is just called directly by f are illustrated in Figure 2.2. Thick lines denote prologue or epilogue code sequences of procedures. To later restart f as if `switch_to_parent` returned to f, we have to (1) capture and save the state of f at the point when f called `switch_to_parent` (C_s in the figure), and (2) transfer control to the return address of f (R_f in the figure), with the values of stack/frame pointers and callee-save registers at the point when P called f (C_f in the figure). The state of f consists of local variables in the frame, incoming parameters of f, and callee-save registers. For saving locals, we need the highest and the lowest address of the local variable save area in the frame of f, and for incoming parameters, we require its size (its offset is a constant through all procedures). Saving callee-save registers is more complex. Since we do not know which callee-save registers `switch_to_parent` destroys, the only feasible way to capture the callee-save registers at C_s is to *actually run the epilogue code of* `switch_to_parent` *and then capture callee*

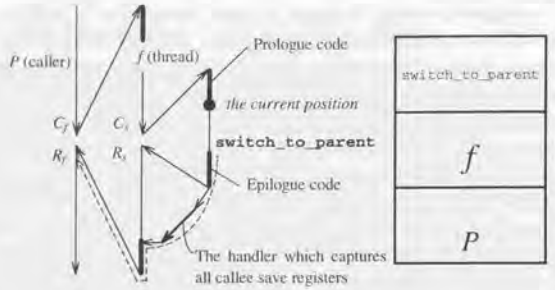


Figure 2.2: Control flow and stack frame layout when P forked f, which called `switch_to_parent` to block f. The control is at the point denoted by the current position and we now resume P. We copy local variables, temporaries, and parameters of f to heap within `switch_to_parent` and capture callee-save registers in a special handler that saves all callee-save registers. Dotted line indicates the control path along which we resume P.

save registers there.[5] We achieve this by modifying the return address of `switch_to_parent` so that it jumps to a special handler routine after running the epilogue of `switch_to_parent`. The special handler routine saves all callee-save registers defined by the register usage convention and then jumps to the epilogue code of f. The epilogue code of f then restores the callee-save registers f uses and returns to P. The control path along which we save callee-save registers for f and resume P is indicated by the dotted line in Figure 2.2. Notice that the control path is equivalent to the regular control path, except that the rest of the procedure body of f is just skipped.

[5]We might now which callee-save registers `switch_to_parent` is using by looking at assembly code generated from it, or by writing it in assembly in the first place. In general, however, `switch_to_parent` is called indirectly from f. In that case, restoring only callee-save registers destroyed by `switch_to_parent` does not restore callee-save registers correctly.

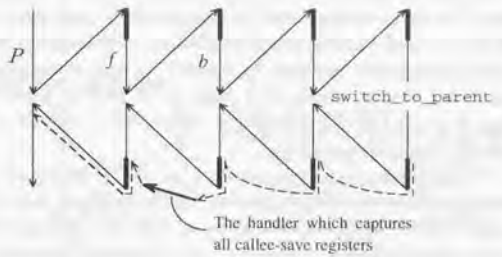


Figure 2.3: Control path along which we resume the current parent of f (general case). Return addresses of all parents but the direct child of f (b in the figure) are redirected to the epilogue code of its parent. The return address of b is redirected to the special handler that saves all callee-save registers.

We have so far assumed that `switch_to_parent` is directly called from f. Let us relax this assumption and now consider a general case where `switch_to_parent` may be called indirectly from a procedure that is called from f. Suppose P forked f, which called a function b, which finally decides to call `switch_to_parent` to block f, perhaps from another procedure which is called from b. The generalized situation is illustrated in Figure 2.3. The semantics we implement is that f later restarts computation *as if b returns to f*. In other words, `switch_to_parent` saves the context of f and resumes P, while *aborting all computation from* `switch_to_parent` *back to f*.

Obviously, the above semantics is rather inconvenient for language implementers. A much more convenient and natural semantics would be that we resume P when f is blocked and we later restart f as if `switch_to_parent` returns to its direct caller. In other words, we save the context of all the call chains between f and `switch_to_parent` and restore all of them when f later restarts. As we will discuss later, this semantics is hard to implement in some procedure calling conventions. Our semantics needs to save/restore only one frame per blocking/resuming.

StackThreads allows `switch_to_parent` to be called indirectly for the sake of language implementers. If it were callable only from the toplevel of the thread itself, a thread must always inline a code sequence that determines whether the thread continues or blocks.[6] In such cases, inlined sequences sometimes become unpleasantly long, so we wish to inline only frequent cases in which a thread can continue with a small overhead and leave other cases in a separate procedure.

To implement the above semantics, we modify the return address of all procedures from `switch_to_parent` back to b. Every frame but b is redirected to the epilogue code of its caller and b is redirected to the handler which saves all callee-save registers. The control flow from `switch_to_parent` to P is threaded through all the epilogue sequences in the call chain, as is indicated by the dotted line in the Figure 2.3.

2.4.4 Resuming a Blocked Thread by Call Chain Reconstruction

Suppose a thread A satisfies the condition whereby a blocked thread f can now restart execution. Thread A can restart f by calling `restart_thread`(c), where c is the heap context filled by `switch_to_parent`. The basic operations are as follows; build local variables and incoming parameter regions for f on top of the stack, restore callee-save registers for f, set FP and SP to the new frame location, and jump to the restarting point. Care must be taken so that, after f finishes or blocks again, f correctly returns to `restart_thread` with the correct callee-save register state. Since we directly jump into the middle of f, f does not run the regular prologue sequence. This in turn implies that executing the epilogue sequence of f does not return to `restart_thread`, but to the *original* caller of f, with values of callee-save registers, SP, and FP for the original caller. Our solution for this consists of two parts. First, `restart_thread` overwrites the slots of f's frame which hold the link to the caller, so that f returns to `restart_thread` with the correct values of FP and SP. More specifically, it overwrites the slots that save the FP of the caller and the return address.

[6] For example, in our implementation of ABCL/f, checking whether or not to block at a potential blocking point includes four conditionals. Such inlined decisions increase code size and obscure the backend C compiler.

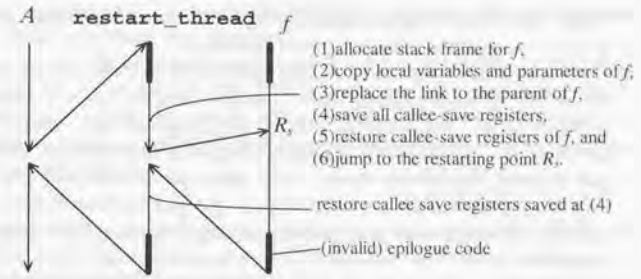


Figure 2.4: Control path along which we restart a blocked thread f. After building a stack frame for f on top of the stack (1 and 2), we replace the return address and the FP of the parent at (3) so that f returns to `restart_thread`. We then save callee-save registers (4), restore callee-save registers captured when f was blocked (5), and jump to the restarting point. The epilogue of f is no longer valid. Hence `restart_thread` restores callee-save registers saved at (4) by itself.

Second, `restart_thread` saves callee-save registers before restarting f, and restores callee-save registers by itself after f returns.

To summarize, restarting a blocked thread involves the following steps: (1) allocate a stack frame for f, (2) copy local variables and incoming parameters onto the stack, (3) replace the link to the parent (return address and the FP of the parent) with the link to `restart_thread`, so that f returns to `restart_thread`, (4) save all callee-save registers, (5) restore all callee-save registers captured when f was blocked, and (6) jump to the restarting point. Since f returns to `restart_thread` with invalid callee-save register state, `restart_thread` restores all callee-save registers saved at (4) after f returned. The control path along which we restart f and f eventually

returns to `restart_thread` is illustrated in Figure 2.4.

2.4.5 Limitations and Discussion

While StackThreads aims to support as wide a range of programming practices in C as possible, there are certain limitations.

First, StackThreads fundamentally relies on the fact that stack frames are mobile, or more precisely, stack- allocated data are accessed relative to the FP or SP. This is not the case for all-C programs and compilers, of course. StackThreads prohibits taking the address of stack-allocated objects and assuming the address remains valid across context switches. Worse, even if a procedure does not explicitly take the address of stack data, an optimizer can cache such an address in a general-purpose register and use the address throughout a procedure. This is done for aggregate data (arrays or structures) allocated on the stack. Overall, StackThreads discourages allocating any aggregate data structure on stack. We believe this is not a fatal restriction for garbage-collected languages.

Second, the reader might realize that there are alternative choices As to which primitives StackThreads supports. It supports saving the context of a *single* stack frame (recall that in Section 2.4.3, when a thread f calls `switch_to_parent` indirectly through a procedure call chain, only the frame of f is saved and other frames in the call chain are simply discarded). A consequence is that a sequential call to a possibly blocking procedure must check a flag after the procedure returns and cascade the unwinding operation if the return value is not present. We investigated the possibility of supporting a sequential call by the library, so that in a sequential call, the control transfer to the caller implies the presence of the return value. We finally concluded this is hard to implement on some calling conventions. To support the above sequential call semantics, the library must unwind multiple stack frames in a single library call and later move them to another location, maintaining the call chain between frames. Moving multiple frames together would be simple if stack frames would be linked with only relative addresses. Unfortunately, however, this is not the casein some procedure linkage conventions including Sparc; each frame stores the *absolute* address of the frame pointer of the caller. To reconstruct call chains in such conventions, each frame would have to supply information about the frame

before making a sequential call. On Sparc, this takes about 10 instructions per frame, nullifying the advantage that the value present check is unnecessary. Our current implementation instead imposes no overhead before a call and the overhead is paid after the call by checking a flag and cascading unwinding operations as necessary.

Finally, StackThreads cannot utilize live variable information to minimize context switch cost. We save and restore the entire region that holds all local variables for a procedure. Although this is potentially a problem, the performance evaluation in Section 2.7 indicates that other factors are more dominating.

2.5 Implementation and Machine Specific Details

2.5.1 General Description

StackThreads exposes the following three interfaces to the programmer.

- `alloc_ctxt(`*desc*, *don't_free*, *add_req*`)`
- `switch_to_parent(`*desc*, *ctxt*`)`
- `restart_thread(`*ctxt*`)`

Desc is a small data structure (*frame descriptor*) which describes the frame format of the direct caller of each procedure and *ctxt* is a pointer to the context returned by `alloc_ctxt`. Set *don't_free?* to 1 to request that the context should be freed by the runtime system before the thread is resumed. Set *add_req* to request additional memory co-allocated with the context. This is provided because it is often the case that the language implementation must allocate a language-specific data structure associated with a context switch. The following discussion ignores these parameters for the sake of simplicity.

A frame descriptor at least contains a field that links itself to the descriptor of the caller. The chain ends at the descriptor of the thread that is about to block. It may also contain some machine dependent fields that provide information hard to achieve from the current procedure. As a design criterion, the overhead for setting up a descriptor should be as small as possible, because a descriptor should be setup when a thread

calls a procedure that potentially blocks the thread. Macros for setting up frame descriptors of a thread and an intermediate procedure between a thread and `alloc_ctxt`/`switch_to_parent` are provided. We call the former `SET_THREAD_DESC` and the later `SET_LINK_DESC`.

We have implemented StackThreads on two platforms, Sparc and Alpha. Below we schematically illustrate each procedure and how it obtains necessary information on each platform, with non-trivial part underlined.

First, `alloc_ctxt` is shown as follows.

```
alloc_ctxt (desc don't_free, add_req)
{
  TL, TH = the lowest/highest address of the thread's frame;
  c = malloc (TH - TL + argument size + add_req);
  initialize c with don't_free, argument_size etc.;
  return  c;
}
```

By traversing the descriptor until the end, we can easily locate the descriptor of the thread. On both platforms, we obtain the argument size from the descriptor of the thread. That is, before the thread makes a call to a procedure that may block the procedure, it sets the size of arguments by an extension supported by GNU C compiler `__builtin_args_info(0)`, which returns the size of parameters of the current procedure.

A less obvious is how to obtain the lowest and highest addresses of the thread's frame. We must be able to traverse the real chain of stack frames in parallel with traversing the chain of descriptor. It turns out this part is very machine specific. We describe details in subsections for each platform.

`switch_to_parent` is schematically shown as follows.

```
switch_to_parent (desc, ctxt)
{
  for each frame f between the current and the grandchild's frame {
      set f's return address to the epilogue of its parent;
  }
  ctxt->ra = the child's return address;
  set the child's return address to the special handler that saves
   all callee save registers;
  dump the thread's frame to ctxt;
  dump the thread's arguments to ctxt;
  /* tell the handler necessary information */
  R = ctxt->ra;
  C = ctxt->callee_save_regs;
  return; /* actually begins epilogue code threading  */
}
```


We modify the return address of every intermediate frame from the current frame up to the frame for the *grandchild* of the thread, to its parent's epilogue code. On both platforms, we obtain the epilogue code of a procedure by an extension supported by GNU C, which allows the address of a label to be used as value. That is, before a thread calls a procedure that potentially blocks the thread, it writes its epilogue code to the descriptor. A code fragment is illustrated in Figure 2.5

After all intermediate frames have been redirected, the child frame's return address is set to a special handler that saves all callee-save registers in ctxt. The handler requires the address where callee save registers should be stored and where to transfer the control after that. They are provided via global variables shown as R and C in the figure. When `switch_to_parent` returns, it actually returns to the epilogue code of its parent, which in turn returns to the epilogue code of its parent, and so on. When the child of the thread returns, it jumps to the special handler. The handler saves all callee-save registers, whose state is now valid for the thread, and returns to its parent, obtaining the address via R. Again, we postpone the discussion as to how to traverse the real frame chain in parallel with descriptor chain and how to find the return address in each frame.

Finally, `restart_thread` is shown below.

```
restart_thread (ctxt)
{
  cs[N];
  frame =  alloca (adequate size for ctxt's args and frame);
  build frame of ctxt in the new frame;
   build args of ctxt in the new frame;
   link the new frame with the current frame;
   copy ctxt->callee_save_regs to cs;
```

```
  ra = ctxt->ra;
  /* ctxt can be freed here */
  restore_proc(new_fp, new_sp, ra, cs);
 RETURN:
  restore callee save registers for itself from cs;
}
```

That is, it expands the stack via alloca by the size that adequately contains both parameter area and the frame of the thread being resumed and restores dumped images on the newly allocated area. So far, nothing is tricky. We then link the new frame to the current frame, so that the resumed thread returns to this procedure. This actually overwrites the return address of the new frame to the address labeled as `RETURN` in the above. The parent FP is modified similarly. Finally we transfer control to the thread, which is done by an assembly routine called `restore_proc`. Before calling `restore_proc`, we copy the dumped callee save registers on stack. It is necessary for being able to free *ctxt* before restarting the caller. `Restore_proc` *swaps* the current contents of the callee save registers and the contents in *cs*, and jump to *ra*, setting SP and FP appropriately. Care must be taken in calling conventions where a procedure may not use FP. In such calling conventions, a procedure may use the frame pointer register for arbitrary purpose. Setting FP 'appropriately' in such calling conventions requires us precisely know if the frame pointer register is used as FP, used as a general purpose register, or not used at all by a procedure. We clarify how to know it on Alpha in Section 2.5.3. What happens when the thread returns? It returns to the label `RETURN`. There, the state of the callee save registers is in general invalid, but FP is valid. FP is valid because we appropriately overwrote the saved frame pointer in the new frame before restarting the thread, thus set appropriately in the epilogue code of the restarting thread. This guarantees that *cs* after `RETURN` is addressed correctly.[7] Here we must clarify how to find the location in the thread's frame where the return address and the frame pointer of the parent are saved.

The discussion so far made it clear that we must be able to:

- traverse the real frame chain in parallel with the descriptor chain.
- locate where the return address is saved in a frame, and

[7] Since `restart_thread` uses `alloca`, *cs* must be addressed via FP.


```
int f (x, y, z) {
  if (x > 0) {
    return ...;
  } else {
    ... = &&EPILOGUE;
    switch_to_parent (...);
  }
 EPILOGUE: ;
}
```

Figure 2.5: The skeleton of a StackThreads procedure. The label `EPILOGUE` is put at the end of the procedure and captured by `&&` operator. `EPILOGUE` refers to the epilogue sequence of the procedure.

- know if a procedure uses FP, and if it does, for what purpose and where the register for FP is saved in the frame for a procedure.

We now clarify machine specific details on Sparc and Alpha.

2.5.2 Sparc

Implementation on Sparc has been done using GNU C compiler version 2.7.2 under the Sparc version 8 calling convention [68]. As the register usage convention, we used the `-mflat` option, which does not use register windows. This convention retains interoperability with legacy C binaries that use register windows.

In `-mflat` convention, non-leaf procedures always setup FP in the register %i7. Hence any non-leaf procedure saves the parent FP in its stack frame. Obtaining the location of them is not straightforward. The stack layout of the `-mflat` convention is shown in Figure 2.6. It locates them at the lowest two words of the local variables save area, putting the caller FP at the bottom and the return address next. The offset of this area from both the FP and the SP varies from one procedure to another. Fortunately, however, the space just below the local variables area is for stack allocation via alloca and we can obtain the address of the bottom word by requesting zero(!) bytes using the alloca (*i.e.*, `alloca(0)`) before making any other

alloca request.[8]

To sum up, before a thread calls a procedure which may eventually call `switch_to_parent`, it writes the result of `alloca(0)` to the frame descriptor. Every intermediate procedure between the thread and `alloc_ctxt` (or `switch_to_parent`) similarly writes the result of `alloca(0)` in its frame descriptor. Having the chain of descriptors each of which contains the location where the parent FP is stored, traversing the real stack frame is straightforward.

2.5.3 Alpha

Alpha uses register $15 for frame pointer. However, the calling convention of Alpha [25, 26] is much more complex than that of Sparc. First, a procedure may not establish a frame pointer. Such procedures access all local variables via SP and may use $15 as a general-purpose register or may not use it at all. Second, even if $15 is used and thus is saved in a frame, its location is not stylized at all. $15 is treated as a regular callee-save register and the location where it is saved depends on all the details of which callee-save registers are used by the procedure. Finally, even if $15 is used as FP, it does not point to the highest address of the current stack frame (stack grows downward). It instead points to the lowest address of the fixed area of the frame. That is, a prologue code of a procedure first expands stack by subtracting the size of the frame from SP, and copies the new SP to $15. SP may be changed by alloca, but $15 is constant through the invocation and is used for accessing local variables hereafter.

The first problem may not be fatal, because one can force a procedure to establish FP, either by a compile option supported both by GNU C and the native C compiler of Digital UNIX, or by calling alloca. The third problem may still not be fatal. Without a pointer to the highest address of the frame of the blocking thread, we cannot precisely determine which portion of the stack should be saved. However, as the last resort, one may be able to overestimate them somehow. Probably, the second problem is the most fatal

[8] The reality is slightly more complex. There may be one word gap between the address returned by `alloca(0)` and the address where the caller FP is saved. In that case, the returned address con-tains nothing. Fortunately, we can distinguish these two cases by reading the one word above the returned address and testing if this word contains the address of a text segment or an address in the stack.

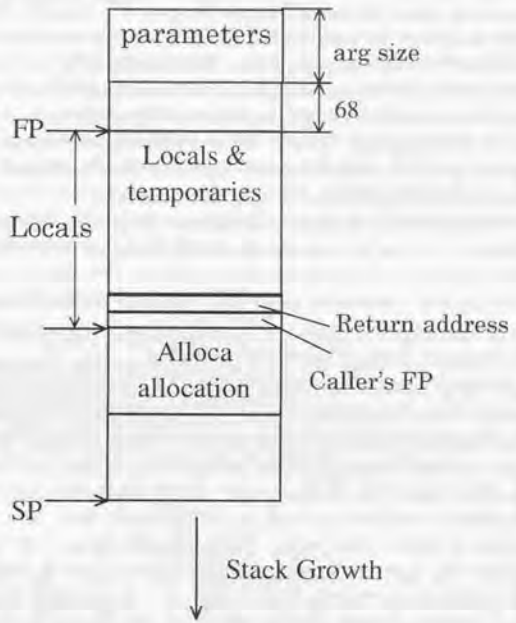


Figure 2.6: Stack frame layout of GNU C Compiler on Sparc with `-mflat` option (ignore register windows). The two words just above the alloca region are the caller's FP and return address.

one. Without knowing where the parent FP is saved, we cannot establish link between the restarting thread and `restart_thread`.

All these features are problematic not only for StackThreads, but also for other language constructs, most notably, exception handling. Operating systems on Alpha (Digital UNIX and Windows NT) supports *runtime procedure descriptor* for implementing this kind of unusual control constructs. It basically provides mapping from a function (actually, any code address within a function) to the descriptor of the stack frame for a function. The descriptor provides various kinds of information for the frame, including its size (of the fixed part), whether or not it establishes FP, which callee-save registers are saved, where the return address is saved, and so on. With such support, everything becomes at least implementable.

Traversing stack in `alloc_ctxt`/`switch_to_parent` does not have any difficulty. It first looks up its own procedure descriptor. It tells us where the return address of the current procedure is saved. We read it and then ask for the procedure descriptor of the caller, using the obtained return address. In this way, we can trivially traverse the real stack frame chain in parallel with the chain of (StackThreads) descriptors.

Setting FP and SP appropriately for restarting a procedure should be done carefully, but possible with runtime procedure descriptor that tells us how does the restarting procedure use $15. If the procedure does not use $15 at all, we do not have to set $15 before restarting it. Since it does not access $15, the current value of $15 does not matter for it, and since its epilogue code does not destroy $15, it will retain the original value, when the control reaches `restart_thread` again. What if it uses $15 as a general-purpose register? $15 is set to the *original* value captured when it blocked. When it later returns to `restart_thread` again, its epilogue code will restore the FP of `restart_thread`. Finally, when if it uses $15 as FP, we set $15 so that it points to the appropriate point in the newly constructed frame.

While this kind of support is very attractive, the cost for associating a code address to its frame descriptor may be significant. Although we have not fully examined the typical cost of the lookup, it seems to take roughly a hundred instructions.

2.6 Implementing Higher Level Abstractions on Top of StackThreads

2.6.1 Remote Read

Figure 2.7 illustrates a simple example in which a procedure suspends its computation when it needs to read remote data. It first checks whether the data is local. If the data is remote, it allocates a context, sends a request message to an owner node, and calls `switch_to_parent` to suspend the procedure. A protocol is defined so that the read datum is put in the global variable R before the procedure is later reactivated. Prior to blocking, a descriptor of the stack frame is filled by `SET_THREAD_DESC(`*td*`)` and passed to `alloc_ctxt` and `switch_to_parent` to carry the information necessary to save and restore the stack frame.

2.6.2 Fork-Join

Consider the simple fork-join protocol illustrated in Figure 2.8. A master thread (master) forks a number of child threads (task) and waits for all the created threads to finish. The master and all the children share a counter (a sync object) which keeps the number of unfinished tasks. The master forks child threads simply by calling the C procedure task in a loop and then calls `join_children` to wait for their completion. Assume each task may block during its computation. The master blocks if some tasks are still unfinished when the master tries `join_children`. There are basically two scenarios. If no children actually block, every procedure call to task simply decrements the counter before returning to the master (line 13). The master then checks the counter in `join_children`, to find the counter is already zero, and falls through (line 42). If, on the other hand, any child is blocked, the counter may not be zero when the master checks it in `join_children`. In that case, the master allocates the heap context, writes the context to the counter, and calls `switch_to_parent` to block itself (line 43–51. When the last child finishes, it will find the context written in the counter and restart the master (line 16–18). This example illustrates how to design and implement synchronizing operations (*i.e.*, operations that may trigger a context switch) on top of StackThreads primitives in a simple case. In this example, `join_children` is the synchronizing operation. In general, a thread which

```
1: {
2:         ...
3:
4:     static struct thread_desc td[1];
5:     /* try to read DATA to x */
6:     if (is_local (data)) {
7:         x = data->val;
8:     } else {
9:         char * c;
10:        SET_THREAD_DESC(td);
11:        c = alloc_ctxt (td);
12:        remote_read_request (data, c);
13:        switch_to_parent (td, c);
14:        /* returns here when unblocked */
15:        x = R->val;
16:    }
17:    /* X has now the right value */
18:
19:        ...
20:
21:  EPILOGUE: ;
22:
```

Figure 2.7: A simple code fragment which blocks the current procedure if data is remote. If remote, it fills a frame descriptor, allocates context, sends a remote read request, and switches to its parent.

calls a synchronizing operation must call SET_THREAD_DESC(*td*), which puts information of the current frame in the area pointed to by *td*, and pass *td* to the synchronizing operation. The synchronizing operation checks the synchronizing condition and when it decides to block, calls SET_LINK_DESC(*lnk*, *td*), which fills the area pointed to by *lnk* with information about the current frame and links *lnk* to *td*.

2.6.3 Return Value Passing Protocols

StackThreads allows a thread to return a value via the normal C return sequence when the thread terminates without blocking. Once a thread blocks, however, it does not make sense for a thread to return a value in this way, because the original caller may not be present just below the current frame. This is inconvenient when building future-like primitive or even a sequential call. Here, we sketch how to express such abstractions in StackThreads, using two previously published schemes as examples.

Concert Hybrid Execution Model

Figure 2.9 shows a variant of lazy context creation scheme in the Concert hybrid execution model [57], changing unimportant details for the presentation. Lazy context creation defines how a potentially blocking procedure passes its result value to the caller, given that control may return to the caller even if the callee has not finished. Suppose procedure f calls procedure g, which may block. If g finishes without blocking, g clears the global flag blocked and returns the result value via C's return statement (line 45–46).[9] If g blocks, on the other hand, g allocates a heap context, sets the flag to point to the context, and switch to f. After control returns to f, f tests the flag and cascades blocking if the flag is non-zero (line16–28). To maintain the call chain between f and g after g blocks, f links g's context to f 's context before blocking (line 23). When g is later resumed and finally finished, g checks if another context is linked from g's context and if one is, resumes it. The return value is written in f 's context (line 47–50).

[9]The original lazy context allocation does the reverse; return value is written into memory, while the flag is returned to the procedure.

```
typedef struct sync {
  int count; /* # of unfinished tasks */
  char * wait;     /* waiting context */
} * sync_t;

void task (s)
    sync_t s;
{
    ... do work. assume we may
        block during computation ...;

    /* I am now finished */
    s->count--;
    /* if everybody has finished and the
       master is waiting, wake up the master */
    if (s->count == 0 && s->wait) {
        restart_thread (s->wait);
    }
  EPILOGUE: ;
}

void master (n)
    int n;
{
    static struct thread_desc td[1];

    /* fork N child tasks */
    sync_t s = make_sync (n);
    for (i = 0; i < n; i++) task (s);

    /* wait for everybody to finish */
    SET_THREAD_DESC(td); join_children (s, td);
    /* continue work ... */

  EPILOGUE: ;
}

/* TD = descriptor of MASTER */
void join_children (s, td)
    sync_t s; thread_desc_t td;
{
    if (s->count > 0) {
        /* when there are unfinished
           tasks, we switch to parent */
        struct thread_desc lnk[1]; char * c;
        SET_LINK_DESC(lnk, td);
        /* allocate context, write it to S,
           and SWITCH_TO_PARENT */
        c = alloc_ctxt (lnk);
        s->wait = c;
        switch_to_parent (lnk, c);
    }
  EPILOGUE: ; }
```

Figure 2.8: A simple fork-join protocol. The sync object (s) counts the number of unfinished tasks. The master blocks on a sync object by leaving the context in it. A finished task decrements the counter and the last task wakes up the master if the master is waiting.


```
typedef struct ctxt
{
  char * ctxt;   /* StackThreads context */
  int result_val;        /* result value */
  struct ctxt * cont;  /* link to caller */
} * ctxt_t;

ctxt_t blocked;
void f ()
{
    static struct thread_desc td[1];
    int r;

    /* code template which calls may-block procedure G */
    r = g ();
    if (blocked) { /* G has blocked, thus F also blocks */
        char * c; ctxt_t f_ctxt;
        ctxt_t g_ctxt = blocked;
        SET_THREAD_DESC(td);
        c = alloc_ctxt (td);
        f_ctxt = make_ctxt (c);
        /* link G's ctxt -> F's ctxt */
        g_ctxt->cont = f_ctxt;
        /* tell the caller of F that F has blocked */
        blocked = f_ctxt;
        switch_to_parent (td, c);
        r = f_ctxt->result_val;
    }
    printf ("result = \%d\n", r);
    ...
  EPILOGUE: ;
}

int g ()
{
    ... computation of G takes place, during which G may block ...;

    /* return sequence of may-block procedure */
    if (g_ctxt->cont) {
        /* G has blocked. Write result value and wake up F. */
        g_ctxt->cont->result_val = result;
        restart_thread (g_ctxt->cont->ctxt);
    } else {
        /* G has never blocked */
        blocked = 0; return result;
    }
  EPILOGUE: ;
}
```

Figure 2.9: A variant of the Concert lazy context allocation scheme. When a procedure g blocks, it allocates a context and sets the global variable blocked to it. After g returns, the caller (f) checks the flag and blocks if it is not zero. The cont field of the g's context is set to the f 's context so that g can later wake up f when g finishes.

First Class Communication Channel Protocol

In the authors' previous work [71], we have proposed an implementation scheme for efficient first-class communication channels. The computation model has no inherent notion of sequential calls. All procedure calls are, at least conceptually, asynchronous calls. Threads communicate and synchronize via communication channels. To return the result value of an invocation, each procedure, in addition to normal parameters, takes an extra parameter, called *the reply channel*. A future call is expressed by a channel creation, a procedure call that passes the new channel as the reply channel, and receiving the result later from the reply channel. A sequential call is just a special case of a future call. Given that thread creation is sufficiently fast, the key question is how to implement channels efficiently.

When a thread passes a new empty channel to a new local thread, it merely sets a special flag value that indicates an empty channel, without allocating memory for it. When the callee writes a value to an empty channel that is not yet assigned to a heap location, it merely writes the value to a register and sets the flag of the channels to indicate it is full. When a procedure terminates leaving one value on the reply channel, it simply returns the value as the return value of the procedure. When a procedure is blocked, on the other hand, the reply channel is converted to a boxed representation—a heap memory is allocated for it and the flag is set to the pointer to it. This boxing operation is also performed when the reply channel escapes from the callee's context; for example when it is passed to a remote processor or stored into a data structure. When the callee tries to receive the result value, it checks the flag and if the flag indicates full, the value is simply extracted as the return value of the procedure. A notable point of this model is its generality and simplicity. There are no inherent notions of sequential calls or even asynchronous calls. Many primitives can be constructed from threads and communication channels, including result value passing and mutual exclusion. Efficiency of frequent cases is preserved by efficient thread creation and the elaborate representation of communication channels.

Chapter 5 describes how this scheme is used in ABCL/f and Appendix B details the implementation.

2.7 Performance Evaluation

2.7.1 Micro Benchmark

In StackThreads, a thread creation is just a procedure call. Thus the overhead for a thread creation is that of a procedure call on the target machine + whatever necessary for implementing a specific language construct. Therefore, the interesting numbers are the cost of blocking and resuming. Table 2.2 and 2.3 breaks down the cost of blocking and resuming in cases where `switch_to_parent` is directly called from the blocked thread. Currently, the cost was measured only on Sparc. Numbers are given as instruction counts on Sparc. The overhead depends on the number of local variables of the procedure (l), the number of incoming parameters of the procedure (P), and the number of callee save registers in the convention (r). The cost of blocking also depends on whether we must allocate a fresh context or can reuse the context of previous blocks of this procedure. In the register usage convention of Sparc where $r = 14$ and a typical procedure where $l = 16$ and $p = 3$, a block costs 267 instructions (when we allocate afresh context) or 179 instructions (when we reuse a context) and a resuming costs 191 instructions. Copying locals and parameters accounts for one third of the total instruction count for a block or a resume. In the simple benchmark program in which a thread repeats blocking and resuming, the time taken for a block-resume pair was 2.31μs on 150Mhz HyperSparc processor. This is comparable to the result reported by Plevyak et al [57]. As is already discussed in Section 2.4.5, our scheme totally ignores live variable information, copying all locals and parameters between stack and heap. Nevertheless, these figures show that this possible limiting factor is not a big problem in practice. Exploiting live variable information would make l (or p) the number of *live* local variables (or parameters) at the point, rather than the *total* number of slots allocated for local variables (or parameters) of the procedure. Even assuming $l = p = 0$ would save at most one third of the blocking or the resuming cost. We also note that almost all instructions for context switch are shared in a library. In the benchmark program, the inlined sequence for blocking is only nine instructions. All other instructions are shared by all context switch sites or are necessary anyhow (such as the epilogue sequence). This is difficult to achieve if we inline context switch

Category	Description	# of Instructions
1. Allocate context	1-1. Calculate context size	33
	1-2. MALLOC	28
	1-3. Initialization	27
2. Switch to parent	2-1. Return address modification	22
	2-2. Copy locals and temporaries	$16 + 3.25l$
	2-3. Copy parameters	$10 + 5p$
	2-4. Set buffer for callee-save registers	11
	2-5. Epilogue	6
3. Save callee-save regs	3-1. Save all callee-save registers	$7 + r$
	3-2. Epilogue of the thread	11[10]
Others		15
Total (with fresh context allocation)		$186 + 3.25l + 5p + r$
Total (without fresh context allocation)		$98 + 3.25l + 5p + r$

Table 2.2: Breakdown of the blocking cost in # of instructions. Parameters l, p, and r refer to the number of locals and temporaries, incoming parameters, and callee save registers, respectively. In 2-4., we set the address of the buffer to a global variable, in which callee save registers are written in 3-1.

sequences to exploit live variable information.

2.7.2 Application Benchmark

Experimental Conditions

We measured performance of three applications listed in Table 2.4, which also shows how threads are forked and switched in each application.

For each application, we first wrote a pure sequential algorithm in C++ and compared it with ones that are augmented with calls to StackThreads primitives. For the evaluation in this chapter, the augmented versions use StackThreads primitives directly from C++ programs, rather than indirectly from ABCL/f. Applications are compiled with GNU C++ (g++) with the highest (-O4) optimization level and all programs run on a single processor (HyperSparc 150Mhz) workstation. There are two augmented versions, one

[9] depends on the thread

Description	# of instructions
1. Setup	21
2. Copy locals and temporaries	$16 + 3.25l$
3. Copy parameters	$10 + 5p$
4. Reinstall caller links	13
5. Copy callee-save registers to stack	$1 + 2r$
6. Check if it should free the context	6
7. Swap callee-save registers with stack	$1 + 2r$
Total	$68 + 3.25l + 5p + 4r$

Table 2.3: Breakdown of the resuming cost in # of instructions. Parameters l, p, and r refer to the number of locals and temporaries, incoming parameters, and callee save registers, respectively.

for evaluating fork overhead and the other for evaluating fork overhead as well as switch overhead. More specifically, we evaluated the following three versions:

SQ: True sequential execution in C++. No multithreading overhead is imposed.

FK: SQ + fork overhead. This version forks threads and checks synchronization conditions, at every point where they would be required in a true parallel/distributed execution. It also forks a new thread at each *sequential* call to a potentially blocking procedure. A potentially blocking procedure is a procedure that may check a synchronization condition in its body. This is necessary because StackThreads does not directly support a sequential call to potentially blocking procedures. The return value from a potentially blocking procedure is obtained via a first class channel protocol presented in Section 2.6.3.

SW: FK + switch overhead. We artificially block the thread at some potential blocking points. The exact conditions in which a thread blocks differ from one application to another.

FK is intended to estimate the overhead imposed on single processor execution of parallel binaries, whereas SW is meant to emulate execution on

parallel processors, assuming zero communication overhead. We assume zero overheads for communication so that switch overhead is not masked by other sources of overhead.

BH simulates motion of particles that interact with each other by Newtonian force. It builds a large tree structure (BH-tree) and force calculation for a particle traverses a part of the tree. A true parallel version partitions particles among processors and processors work in parallel. A thread blocks when it accesses a node of a BH-tree that is not present locally. SW version emulates this behavior on a single processor, by blocking a thread when it accesses a node that has not been accessed 'recently.' When a force calculation starts, no BH-nodes have been accessed recently. When a node is accessed, the computation is blocked and the node is marked "recently accessed." Once a node is marked, accesses to the same node do not cause further blocking. We clear all the marks every 128 particle. This emulates an execution where each processor is responsible for 128 particles. Since each recursive call to a BH-node potentially encounters a remote node, each recursive call is a potentially blocking procedure call. Hence we fork a thread at each recursive call both in FK version and SW version.

RNA is a combinatorial tree search problem with pruning. A true parallel program extracts parallelism simply by making recursive calls in parallel when the recursion depth $< D$, where D is given at command line. The processor on which a parallel recursion is invoked is determined randomly. A thread blocks to wait for the completion of recursive calls. SW version of RNA emulates this behavior by blocking a thread at every recursive call whose recursion depth $< D$. In the true distributed memory execution, the caller of a remote call is not blocked if it has other works when the remote processor is evaluating the remote call. Thus, this pessimistically emulates execution on a true distributed memory machine in terms of switch frequency. A thread is forked at each recursive call both in FK version and SW version, regardless of the depth.

CKY is a parser for context free grammars. Given an input sentence of length n, it calculates $1/2\ n(n+1)$ sets of symbols, which we denote as $S_{i,j}$. $(0 \leq i < j \leq n)$. A true parallel version forks a separate thread for each $S_{i,j}$ and distributes them on processors. The thread which computes $S_{i,j}$ requires $S_{i,k}$ and $S_{k,j}$ for any k $(i < k < j)$ and blocks if data has not been produced when necessary.

Application	**FK/SW** fork a thread at:	**SW** switches a thread when:
BH	each visit at BH tree node	the node is not accessed recently
RNA	each visit to a node in the search tree	the depth of the node $< D$ (D is a constant)
CKY	each computation of $S_{i,j}$	$S_{i,j}$ accesses $S_{i,k}$ or $S_{k,j}$ before computed

Table 2.4: Benchmark Applications

SQ version of CKY calculates $S_{i,j}$ $(0 \leq i < j \leq n)$, one after another, in an order that $S_{i,j}$ with smaller $j-i$ are computed before $S_{i,j}$ with larger $j-i$. This naturally guarantees that necessary data has already been produced when necessary. FK attaches a thread for each computation of $S_{i,j}$. SW *reverses* the order in which $S_{i,j}$ are computed. This scheduling order blocks all threads with $j - i > 1$ at least once.

Benchmark Results

Figure 2.10 shows the execution time (relative to true sequential execution) of the FK and SW version, on Sparc and Alpha. FK estimates the overhead that appears when we execute the parallel program on a single processor. The sources of overhead include forks, synchronization condition checks, and creation of synchronizing data structures. SW estimates the fork and switch overheads that appear in true parallel execution on distributed memory parallel computers (other sources of overheads such as communication overhead is not included). In all the benchmarks, FK overhead is within 15% and SW overhead is within 30%. In CKY, the fork overhead is relatively high because accesses to $S_{i,k}$ and $S_{k,j}$ by the thread which computes $S_{i,j}$, which are just an array reference in true sequential version, are performed by synchronizing accesses. The overhead also includes creation of synchronizing data structure for each $S_{i,j}$.

Table 2.6 and Table 2.7 show the execution time of each version (call them T_{SQ}, T_{FK}, and T_{SW}, respectively) in milliseconds on Sparc and Alpha, respectively. Table 2.5 shows the number of forks (F), the number of synchronizations (S), and the number of blocking (B) which occurred in the

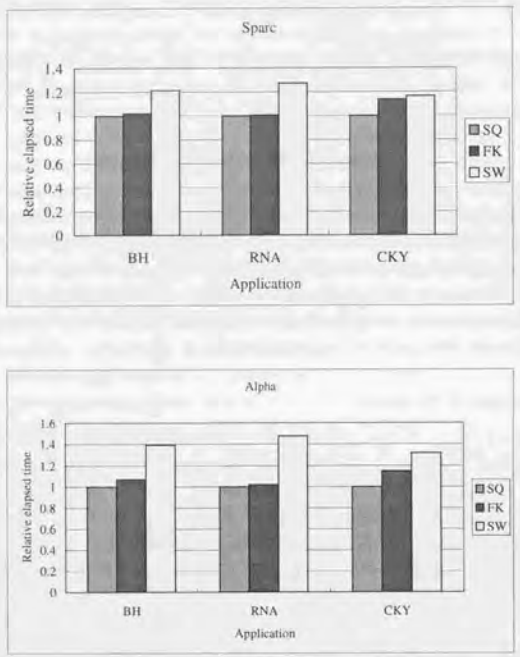


Figure 2.10: Overhead of each version relative to true sequential program on Sparc and Alpha.

Application	No. of forks (F)	No. of syncs (S)	No. of blocks (B)
BH	1,298,124	1,323,554	68,360
RNA	160,411	177,341	33,861
CKY	14,560	941,690	28,524

Table 2.5: Number of forks in FK/SW (F), synchronization or potential blocking points in FK/SW (S), and actual blocks in SW (B).

Application	T_{SQ}	T_{FK}	T_{SW}
BH	5,224	5,321	6,338
RNA	1,095	1,101	1,395
CKY	5,803	6,602	6,746

Table 2.6: Execution time for each version (T_{SQ}, T_{FK}, and T_{SW}) in milliseconds (on Sparc).

Application	T_{SQ}	T_{FK}	T_{SW}
BH	1,016	1,126	2,351
RNA	371	398	570
CKY	3,438	3,664	3,869

Table 2.7: Execution time for each version (T_{SQ}, T_{FK}, and T_{SW}) in milliseconds (on Alpha).

benchmark. In other words, S is the number of potentially blocking points, and B the number of actual context switches. Execution times are given in milliseconds on HyperSparc 150Mhz processor and Alpha 333Mhz processor. As expected, the cost of switch is somewhat higher on Alpha.

$(T_{SW} - T_{FK})/B$ gives us a rough approximation of the cost of a switch. Although they vary depending on application they are roughly from $5\mu s$ to $15\mu s$ on Sparc and from $5\mu s$ to $18\mu s$ on Alpha. We have not fully analyzed the source of the different switch cost in the applications.

2.8 Summary

StackThreads offers a practical approach to implementing efficient multithreading languages. It supports very efficient thread creation and thread switching between normally written C procedures. Unlike previous implementations of efficient multithreading, it does not require extensive cooperation from the code generator. Thus, the compiler writer can map the sequential constructs of the language straightforwardly onto C, while using the low overhead multithreading support of StackThreads for parallel/concurrent primitives. Performance measurement through three parallel applications shows encouraging results on Sparc. Overheads for fork and synchronization checks are never significant ($< 15\%$). Switch cost is comparable to one of the best-known results [57] which, unlike ours, needs a cooperating compiler. Finally, switch frequency in these applications is low enough to make the overhead of switches acceptable in practice ($< 30\%$).

Chapter 3

Garbage Collection

This chapter describes the design and implementation of a garbage collection scheme on large-scale distributed-memory computers and reports various experimental results. The collector is based on the conservative GC library by Boehm & Weiser. Each processor traces local pointers using the GC library while traversing remote pointers by exchanging "mark messages" between processors. It exhibits a promising performance. In the most space-intensive settings we tested, the total collection overhead ranges from 5% up to 15% of the application running time (excluding idle time). We not only examine basic performance figures such as the total overhead or latency of a global collection, but also demonstrate how local collection *scheduling* strategies affect application performance. In our collector, a local collection is scheduled either *independently* or *synchronously*. Experimental results show that the benefit of independent local collections has been overstated in the literature. Independent local collections slowed application performance to 40%, by increasing the average communication latency. Synchronized local collections exhibit much more robust performance characteristics than independent local collections and the overhead for global synchronization is not significant. Furthermore, we show that an adaptive collection scheduler can select the appropriate local collection strategy based on the application's behavior.

3.1 Introduction

Although many high-performance parallel programming languages and their implementation techniques have been studied, the performance of garbage collection on large-scale parallel machines is not yet well understood. In particular, performance studies of garbage collection on distributed-memory parallel computers are rare, presumably because of the complexity of implementation. This chapter describes the design and implementation of a garbage collector for large-scale distributed-memory parallel computers and examines its performance.

We extended Boehm & Weiser's conservative garbage collection library [15, 16] to distributed-memory parallel machines. Our collector preserves the spirit and advantages of Boehm & Weiser's GC including the capability of working with C and C++ programs and a modest heap expansion policy. The collector is used as part of the runtime system for a concurrent object-oriented language ABCL/f.

Like other collectors, our collector consists of two levels—*local* and *global*. In addition, it has two kinds of local collection, namely, *independent* (or *asynchronous*) and *synchronized*. An *independent* local collection is a collection in which a collecting processor independently reclaims its local garbage without any coordination with other processors. A *synchronized* local collection is a collection in which all processors perform a local collection at the same time. These two local collections differ only in how they are scheduled. An independent collection is invoked without any notification being sent to other processors, while for a synchronized one, notification is sent to a master processor, which requests all other processors to do a local collection. A *global* collection is a simple distributed marking collection in which all processors cooperatively traverse the entire object graph, exchanging "mark messages" to trace remote pointers.

We investigate performance characteristics of the collector using four applications that exhibit various allocation and communication behaviors. Conclusions derived from the experiments include:

- The cost of barrier synchronization for synchronized local collection and global collection is insignificant, at least in our experimental environments (up to 256 processors). They are implemented with a simple1-to-N communication and can certainly be improved.

- Several simple techniques significantly reduce the overhead of the distributed marking collection. With these techniques, the overhead of global collections becomes insignificant. Global collections occupy from 5% to 30% of the total running time of the application *excluding idle time due to load imbalance and communication*. In the most space-intensive setting in the experiments, they occupy at most 15%.

- Application performance is affected not only by allocation/collection performance *per se*, but also by how collections are scheduled. In particular, synchronous applications, which frequently use synchronous communication and have little intra-node parallelism, are very sensitive to the scheduling skew introduced by independent local collections.

- We can adaptively select the appropriate local collection strategy by examining application behavior. The adaptive scheduler synchronizes local collections by default and uses independent collections when it presumes that the application tolerates long communication latency.

The rest of the chapter is organized as follows. Section 3.2 reviews previous work on collection schemes in parallel and distributed environments. Section 3.3 describes the design and implementation of our collector. Section 3.4 devotes to the collection selection and heap expansion policies of our collector. Section 3.5 describes the experimental conditions. Section 3.6 shows the overall collection overhead. Section 3.7 demonstrates the importance of collection scheduling strategies. After discussing alternatives and limitations of the work in Section 3.8, we summarize this chapter in Section 3.9.

3.2 Related Work

Many proposed collection schemes on parallel and distributed systems are multilevel. They typically consist of local collections and a global collection and local collections are typically scheduled independently. On distributed-memory machines, global collection schemes are roughly classified into two categories, which are reference counting schemes and distributed-marking schemes [1, 39, 56]. Reference counting schemes keep track of how many references exist for each object and delete objects to which there are no references. Distributed-marking schemes are natural extensions to local-marking

collectors. They traverse the entire object graph, which spans multiple processors, exchanging "mark messages" to trace remote references.

3.2.1 Local Collection + Reference Counting

There have been many proposals that extend reference counting schemes to distributed-memory machines [10, 11, 30, 55, 77]. Implementations of distributed programming languages typically favor reference counting [12] and some parallel languages adopt them [38, 60]. Since managing reference counts on every pointer duplication and deletion incurs very expensive overhead, reference counting in distributed environments is often combined with a local tracing collector and keep track of reference counts only for *remote* references. Each processor independently performs local collections and the local collector counts how many remote references the processor still holds for each remote object [11, 38, 46].

It is still an open question under what circumstances distributed-marking collectors perform better than reference counting collectors, and this thesis does not address that issue. Here, we just make a few remarks that contrasts distributed-marking and reference counting.

- Reference counting collectors send messages (so called *delete* messages) along 'dead' edges of the global object graph (*i.e.*, references that point to dead objects), whereas distributed-marking collectors send messages (so called *mark* messages) along 'live' ones. Therefore, as in uniprocessor collectors, whether or not one scheme is more favorable than the other depends on the live/dead ratio; distributed-marking collectors will be favorable when the percentage of live objects in heap is relatively small.

- In typical reference counting collectors, each processor triggers local collection *on its own initiative*, followed by sending delete messages along remote references no longer used by the processor. These delete messages contribute to reclaiming space in the next local collection at the remote processors. In other words, a processor performs a local collection when *that processor* runs out of space, but whether or not the collection is successful depends on how many delete messages reached that processor, or, how many local collections have occurred in *other* processors that send delete messages to that processor. This form of 'uncooperative' local collection scheduling, which we believe is typical in state-of-the-art systems, may unnecessarily delay reclamation of garbage that would otherwise be collected more promptly. For example, reclamation of a data structure that spans N processors must wait for those N processors to perform their local collections on their own initiative. A data structure that remains unnecessarily long degrades local collector performance, which in turn affects the overall performance significantly.

- Our observation that independent local collections often degrade performance of synchronous applications (as will be presented in Section 3.7) should apply to reference counting schemes as well.[1] This partially nullifies the conventional wisdom that an advantage of reference counting is that it allows processors to collect independently.

3.2.2 Distributed Marking

Many algorithms based on distributed marking have been proposed [37, 40, 56]. Since they have been studied mainly in the context of loosely coupled distributed environments, attention has mainly been focused on how to avoid barrier-synchronization or tight coordination between processors. Most of them are complex in order to overcome problems in distributed environments such as faulty processes or lost messages, and have not been implemented. There has been little work in the context of parallel high-performance computing. One study [41] focuses on superficial aspects such as the number of messages or the total overhead of global collections. How they affect overall application performance has not yet been studied. Compared to the many proposed algorithms, our global collector is rather simple. Our global collection assumes reliable message delivery, no faulty processors, and no concurrency with user programs. We are interested in the performance of such collectors and how such global collections should be combined with local collections to achieve overall application performance.

[1] The same observation has been reached by others, although not published; implementation of the KL1 on PIM [38] uses reference counting and provides a primitive by which the programmer can explicitly trigger local GCs on all processors. The primitive was found to be useful by application writers [21].

3.3 Design and Implementation of the Collection Scheme

3.3.1 Overall Design

Our collector defines *local* and *global* collections, and local collections are further classified into *synchronized* and *independent* collections. We reuse the basic functionality of the GC library such as heap management, pointer identification, local pointer traversing, and local allocation. The heart of our extension is (1) how to adapt it to distributed-memory parallel computers where an object may be referenced from other processors and (2) when we should invoke which collections. In the following subsections, we first briefly summarize relevant information about the GC library and then describe our extensions.

3.3.2 Boehm & Weiser's GC Library

Boehm & Weiser's conservative GC library [15, 16] is a garbage collector that can work with C and C++ programs. An important design goal is to minimize cooperation required from application programs. The programmer simply calls `GC_MALLOC(`*size*`)`, instead of `malloc`, to allocate *size* bytes of memory. Unlike `malloc` in the C standard library, the programmer does not have to explicitly free the allocated memory. The garbage collector automatically reuses blocks of memory that are no longer being used. Since the runtime data structure of C programs does not have enough information that precisely distinguishes pointers from non-pointers, the collector conservatively assumes that any value that appears to be a pointer is in fact a pointer.

Omitting irrelevant details, free memory is managed via free lists that are segregated by their object sizes. When an allocation request is made, the allocator tries to find a block of memory from the appropriate free list. When an allocation request cannot be met from the current heap, the allocator either invokes a garbage collection or expands the heap by requesting memory from the operating system. It invokes a garbage collection if the application has allocated enough memory since the last collection. This places an upper bound of the collection frequency, as long as the request to the operating system is successfully served. The user can customize the threshold value

that determines whether to invoke a garbage collection. More precisely, letting H be the current heap size, A the amount of memory allocated since the last collection, and r the customizable parameter, the allocator decides to invoke a collection when:

$$H/r < A.$$

That is, a collection is invoked when at least one rth $(1/r)$ of the current heap size has been allocated since the last collection. Smaller values of r tend to produce less frequent collections at the expense of space.

To understand the impact of parameter r, let us estimate the heap size for an application, assuming a very simple allocation behavior and a lifetime distribution. We assume the application holds L bytes of long-lived objects and continues allocating short-lived objects. By "long-lived" objects, we mean objects whose lifetime is beyond the typical collection interval. In this case, the collector tries to keep the size of the heap (H) at

$$H = \frac{r}{r-1}L$$

The application behavior in this state is to repeat the allocation of $H/r = L/(r-1)$ bytes, followed by a collection which retains L bytes while freeing $H - L = L/(r-1)$ bytes for the next allocation cycle. For example, when r is minimum (*i.e.*, $r = 2$),[2] the heap is expanded to $2L$ bytes, leaving L bytes for allocating short-lived objects. When $r = 4$, which is the default setting, the heap is expanded to $1.33L$ bytes, leaving $0.33L$ bytes for allocating short-lived objects. Notice that the heap usage of the collector is much more modest than that of typical collectors used in implementations of heap-intensive programming languages such as SML/NJ [3].

Our extension preserves the simple interface to C and C++ programs and the modest heap expansion policy of Boehm & Weiser's GC.

3.3.3 Representation and Management of Remote References

Exit Table

A reference to a remote object is represented by a pointer to a special type of object called a *stub* (a.k.a. *proxy*). A stub for an object remembers the processor number and the address of the object body.

[2] By definition, $r = 1$ effectively inhibits garbage collection. The value for r can only be specified as an integer. Thus the minimum value for r is 2.

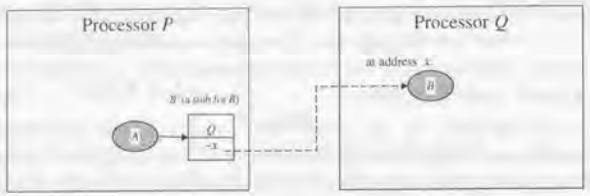


Figure 3.1: Object A on processor P references object B allocated at address x on processor Q. A actually points to stub B' which is allocated on P. Stub B' holds processor number Q and the bit-wise negation of x.

A stub actually holds the bit-wise negation of the address of the object body, rather than the address itself, so that the local conservative collector does not misidentify the address as valid in the local processor.[3] Figure 3.1 illustrates a situation where an object A on processor P references another object B, which was created by processor Q at address x.

All stubs in a processor are registered in a hash table called an *exit table*. When a processor receives a reference to a remote object for the first time, the receiver registers the object in the exit table of the receiver processor. Upon subsequent receptions of the same address, the receiver looks up the address in the exit table to avoid having multiple entries for the same address.

Entry Table

To enable local collections, each processor keeps track of objects that are created by the processor and may still be referenced from other processors. Such objects are registered in a table called an *entry table*. When a processor sends a reference to an object to another processor for the first time, the object is registered to the entry table of the sender processor. Once

[3]Let H be the maximum heap address, M the greatest possible pointer value, and x any valid heap address. If $H < M/2$, as is the case in most address space configurations, bitwise negation of x $(= M - x)$ is larger than $M - M/2 = M/2$, hence is not a valid heap address. The idea is borrowed from the finalization code of Boehm & Weiser's GC.

registered, the object is never reclaimed without cooperation from other processors. The local collector simply regards the entry table as a root of a local collection, so that objects that are referenced from other processors are retained. In our collector, references can be removed from an entry table only by a global collection.

All objects that may be referenced from other processors as well as all stubs have the following common header fields:

Flag: A flag that distinguishes the type and the status of the object. The flag is either a stub, a body of an object registered in the entry table, or a body of an object not registered in the entry table.

Export Count: The number of messages that contain a reference to the object and have not been delivered to the application program.

The runtime system checks the first word of an object to see whether the object is a stub or the body of an object. If the object is not a stub, the flag also indicates whether the object is registered in the entry table. The role of the export count will be described in Section 3.3.3

We currently store these items of information in the header of an object. Thus, every object that may be referenced from other processors as well as every stub must conform to this format. A better interface to C and C++ applications would be to allocate them in a separate space so that programmers would not have to be aware of the format. We chose to embed the header in an object just for simplicity of implementation and for fast access to the headers.

Obtaining Consistent Snapshots

In distributed-memory computers where references to objects are carried by messages, the garbage collector must traverse a globally consistent snapshot of the object graph. The definition of a snapshot includes messages that have not yet been delivered to the application program [18]. In other words, the garbage collector must somehow find references in undelivered messages. An implementation of a garbage collector could achieve this by examining the message buffer. This approach however requires the message format used in the application be known to the collector, reducing both the reusability and the flexibility. Alternatively, the collector could run concurrently with

the application and traverse a message when the application unpacks it. This also requires a very tight coordination between the application and the collector. The application must notify the collector whenever it unpacks a message. Another problem with this approach is that it is difficult to guarantee that the collector will make enough progress.

The approach we took in our collector reduces the cooperation from the application and simplifies the interface to the collector. As shown in Section 3.3.3, all objects as well as stubs have a counter that stores the export count. When an application sends a reference to an object to another processor, it increments the export count of the object. Conversely, when an application receives a reference to an object from another processor, it decrements it. These operations are local—the creator of an object operates on the body of the object, while other processors operate on their local stubs for the object. The invariant is that the global summation of the export counts over the body and all stubs for an object equal to the number of messages that have been issued by a sender but have not been delivered to the receiver. Before a global collection, all processors synchronize and retain all objects whose export count is positive. This can be implemented *on top of* any send/receive-type message-passing interface and makes the communication layer and garbage collector independent. The interface to the application is also quite simple. The application simply calls `export(`o`)` when it sends a reference to an object o and calls `import(`p, o`)` when it receives a reference to an object o that is allocated at processor p. These procedures increment and decrement the export counts. As long as this cooperation is given to the collector, the application freely chooses its message format, buffering policy, flow control, and so forth.

One expense of this mechanism is that all processors must synchronize and calculate the global summation of export counts. We report the overhead of this process in Section 3.6.

3.3.4 Collection Algorithms

Local Collections

When a processor decides to invoke an independent local collection, it simply invokes a local collection, regarding the entry table of the processor as a root. It is a completely local operation.

When a processor decides to invoke a synchronized local collection, on the other hand, it sends a request to a master processor. The master arbitrates almost simultaneous collection requests from other processors and broadcasts a request message to all the processors. When a processor receives the message, it performs a local collection and then continues. This way, a synchronized local collection is implemented with a single request plus a broadcast.

Global Collections

When a processor decides to invoke a global collection, it sends a request to the master processor, just as in a synchronized local collection. After arbitrating simultaneous requests, the master controls the progress of a global collection in the following steps.

Sync phase: stops the user program,

Undelivered phase: finds objects that are referenced from undelivered messages,

Mark phase: marks objects starting from local roots of all processors, and

Finish phase: finishes a collection.

The *sync* phase guarantees that, after this phase, user-level activities are stopped and no user-level messages are being transmitted in the network. This is done as follows. For each pair of processors (P, Q) $(P \neq Q)$, processor P maintains two counters which count the number of user-level messages sent by P to Q as well as those received by P from Q. That is, each processor maintains $2(p-1)$ counters where p is the number of processors. Each processor requests all other processors to return the number of received messages from that processor and stops delivering incoming messages to the application. These received, but undelivered messages are buffered and delivered to the application after the current global collection. Each processor signals the end of this phase when all the user-level messages have reached the receiver processor. Notice that this does not mean that messages have been *delivered* to the application. It simply means that all the user-level messages have been delivered to the application *or* buffered by the collec-

tor running at the receiver processor. After this phase, the global collector exclusively receives GC-related messages.[4]

The *undelivered* phase finds objects referenced from undelivered messages, which are buffered during the previous phase. As mentioned in Section 3.3.3, we achieve this without examining the buffered messages directly, by maintaining the export count for each object. Each processor scans its exit table, reads the counter of every stub, sends counters to owner processors, and zeros the counters of stubs. Each processor then accumulates received counts into object bodies. Objects whose exported count is zero are the subject of the current global collection. Other objects, whose export count is positive, are simply retained. When this exchange has been done, each processor clears entries in the table for objects that are the subject of the collection.

The *mark* phase begins marking from the root in each processor. This phase proceeds as follows. Each processor initiates marking from its local root. When the marking is finished, each processor finds stubs that are marked during the marking and sends "mark messages" to owner processors. Upon receiving a mark message, the receiver processor registers objects in the mark message to the entry table. The processor then resumes local marking from these newly registered objects.

In this way, each processor continues local marking as far as possible and then performs remote marking from all the stubs that were marked during the previous local marking. In this way, senders reduce the message overhead by requesting several items of work with a single message. In addition, we found a similar buffering policy at the receiver side equally important for further reducing the message exchange overhead. Upon reception of a mark message, the receiver buffers the mark message and tries to receive further mark messages. A processor buffers mark messages until the buffer for mark messages becomes full or until there are no incoming messages in the network. Figure 3.2 illustrates the main loop of this phase.

An acknowledgment for a mark message is returned when all objects reachable from the mark message have been marked. The mark phase finishes when all the processors have received all the acknowledgments for their

[4]This phase is unnecessary if the user-level program can be separated from GC messages, as is the case when, for example, the user-level program and the collector use separate message buffers.

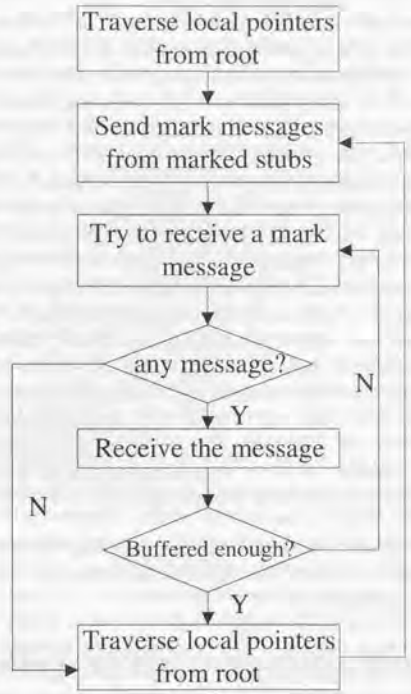


Figure 3.2: The main loop of the mark phase of a global collection. Each processor traverses local pointers as far as possible, finds marked stubs in the local traversal, and sends mark messages to other processors. Mark messages from other processors are buffered until the buffer is full or until no messages are found from the network.

first remote marking.

There are some subtle implementation issues. First, since Boehm & Weiser's GC recognizes the exit table as a root, simply invoking a local collection marks all the stubs in the first local marking. Therefore, we need to deceive the local collector. Before marking from the local root, each processor overwrites all the pointers from the exit table to stubs with bit-wise-negations of these pointers. This way, we effectively hide all the stubs from the local collector, while correctly retaining the structure of the exit table. These pointers are restored when the first local marking is finished. Second, it is inefficient to scan the entire exit table after every single local marking. We need a way of quickly finding objects that are marked during a local marking. This is a tricky task again because the local collector has no special knowledge about stubs. We solve this problem by using a clever data structure for the exit table. Stubs in the exit table are grouped into several bins. Stubs in a bin all share a one-word object. When the one-word object for a bin is not marked after a local marking, there is no possibility that any object in the bin has been marked, so we can skip all the stubs in that bin. Mark bits for the one-word objects are cleared after every local marking. We found this technique very important. This was particularly important, particularly because second or later local markings actually mark few stubs. That is, most live objects are reachable from the root with a small number of remote links.

Finally, the *finish* phase finishes collection by reclaiming unmarked objects. Boehm & Weiser's GC defers the reconstruction of the free lists until an allocation request demands it.

3.4 Collection Scheduling and Heap Expansion Policies

3.4.1 Problem Statement

When an allocation request cannot be served from the current heap, the allocator has three choices, namely, a local collection, a global collection, or a heap expansion. There are further choices as to which local collection should be invoked, namely, independent or synchronized. The goal is to avoid unnecessarily frequent garbage collections with a reasonable heap-size.

Figure 3.3 illustrates our policy that is explained below.

Remark: We note that the goals of most current garbage collectors, including Boehm & Weiser's GC and ours, are *not* to maximize *speed*, but to achieve a reasonable speed with a reasonable space. As a matter of fact, the most aggressive policy that expands the heap until it exhausts a large fraction of the physical memory would achieve a nearly optimal speed in most cases (at least for small applications that are unlikely to exhaust the physical memory, and thus unlikely to be swapped out).[5] Most collectors do not behave in this way, however. This is partially because it is difficult to correctly capture the tradeoff between time and space—we can save time by using more space in normal circumstances, but such policy occasionally incurs a large cost in events which occur unpredictably (*e.g.*, swap out). Thus the primary policy taken in most collectors is to play safe by keeping the heap size reasonable and expanding it only when there is a strong motivation to do so (*i.e.*, when collections would otherwise become too frequent). Our policy as to when and which collections should be invoked is based on the same policy.

3.4.2 Local Collections

As mentioned in Section 3.3.2, when an allocation cannot be served from the current heap, Boehm & Weiser's GC invokes a collection if the application has allocated at least H/r bytes since the last collection, where H is the heap size and r a parameter chosen by the user. A natural adaptation of this policy to distributed-memory parallel computers would be to invoke a local collection when at least H/r bytes have been allocated on a processor since the last local collection, where H is the local heap size of the processor.

We slightly modified this policy to take *synchronized* local collections into account. Each processor is informed of the largest local heap size over all the processors. Let us call the value M, which is made consistent at every global collection. When a processor's local heap size is smaller than cM where c is a constant close to 1.0 (we currently set c to 0.8), the processor expands the heap without invoking a garbage collection. In other words,

[5] In particular, this is almost always the case in single-task environments such as the default operating system for AP1000+.

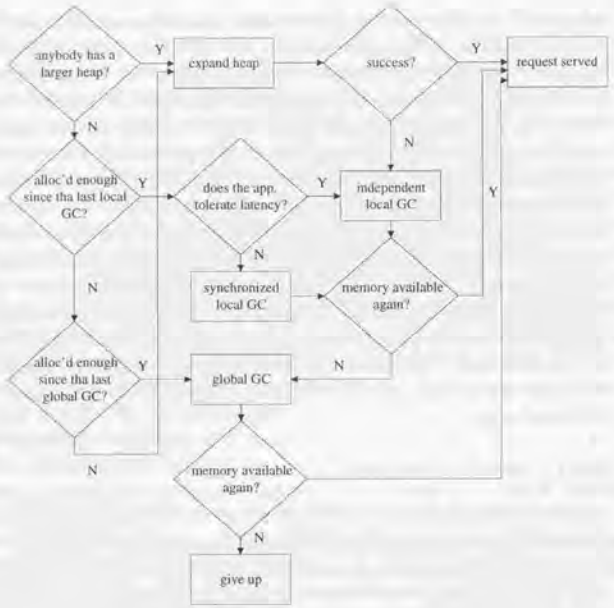


Figure 3.3: The decision diagram when an allocation cannot be served. It unconditionally expands the heap to around the maximum heap size among processors. If enough allocation has been done since the last local GC, it invokes a local GC. Or if enough allocation has been done since the last global GC, it invokes a global GC. Otherwise it tries to expand the heap. Whether or not to synchronize local GC is determined by the criteria explained in the main text.

a processor invokes a local collection when at least H/r bytes have been allocated on that processor since the last local collection *and* its local heap size is close to the maximum heap size among all the processors. This policy applies both to synchronized and independent local collections.

The policy is justified based on the following observations. If any processor has already expanded its heap size to M bytes, it is reasonable for other processors also to expand their local heaps to around M bytes. In other words, if any processor requires M bytes, it is reasonable in distributed-memory computers to expand the *total* heap size to around nM bytes, where n is the number of processors. In fact, if any processor requires M bytes, other processors are, sooner or later, *likely* to require roughly the same amount of space. Thus if this policy would not be taken, synchronized local collections would become unreasonably frequent. If *any* processor is willing to do a synchronized local collection, other processors are also forced to do so. The net effect would be that the frequency of synchronized local collections becomes the maximum local collection frequency over all the processors. We avoid this effect simply by allowing any processor's local heap size to be expanded without garbage collection until it catches up to any other processor's local heap size.

3.4.3 Global Collections

A global collection is invoked when a local collection is not invoked by the criterion in the previous section and when enough allocation has been done since the last global collection. More precisely, letting H be the local heap size, A_l the amount of allocation done since the last local collection, A_g the amount of allocation done since the last global collection, and r a customizable parameter, we invoke a global collection when:

$$A_l \leq H/r < A_g.$$

That is, when enough allocation has not been done since the last *local* collection, but has been done since the last *global* collection.

3.4.4 Choices between the Two Local Collections

When a processor decides to invoke a local collection, it must then decide whether the collection should be independent or synchronized. The most

important point is that processors that are collecting do not respond to requests from other processors. Thus a request to a locally collecting processor is delayed until the local collection is finished, increasing the latency of the communication. As we will see in Section 3.7, this sometimes affects application performance significantly.

Thus the criteria of the local collection scheduler are:

- Synchronize by default (at startup or uncertain cases)
- Switch to independent mode when the system is sure that the program is latency tolerant
- Quickly recover from the independent mode if the decision turns out to be wrong.

The main problem lies in the second item—how to detect if an application, currently running in synchronized mode, is actually latency-tolerant. We achieve this by examining how many times each processor enters the idle state during an interval. From the number of idle state periods, each processor calculates how performance would be degraded if idle periods would be made longer by the receivers' local collections. Each processor reports this number and its processor utilization to the master. The master presumes that the application is currently latency-tolerant if most processors report a small degradation factor.

More precisely, suppose the system is currently in the synchronized mode and we are about to start a synchronized local collection. For each processor, let L be the interval between the last synchronized collection and the current one and n be the number of times the processor became idle during that interval. We define the *degradation factor* (D) of this interval by:

$$D = \frac{npg}{2(L+g)}$$

where G is the estimated time of a single local collection and $p = G/(L + G)$, which approximates the probability that a given communication would be delayed by a local collection on the destination processor. Given the estimated single local collection time, G, the average additional delay when the receiver processor happens to be locally collecting is $G/2$. By multiplying np and $G/2$, we obtain the total additional delay that would have been

imposed if the last interval were in the independent mode. To sum up, D represents how much the last interval would have been affected if it were in the independent mode. Together with D, processor utilization is also collected in the master. The master presumes that independent collection is more desirable if:

- the number of processors that report D larger than a threshold (= 0.3 in the current implementation) is less than a threshold (= one eighth of the total number of processors), and
- the number of processors that report utilization smaller than a threshold (= 0.5) is less than a threshold (= half the total number of processors).

The first condition says that few processors would be damaged by the increased average latency, hence the application will be latency-tolerant. The second condition rules out cases where many processors are idle, so having global synchronization on each local collection unlikely to matter. In such cases, we prefer to insist on synchronized local collection for safety.

When the system is running in independent mode, on the other hand, each processor individually checks its processor utilization after each independent collection. If utilization is lower than a threshold (= 50%) the processor notifies the master. The master decides to revert to the synchronized mode when a predetermined number (= one sixteenth of the total number of processors) of notifications have accumulated.

The actual state transition is controlled by a three-state ((1, 0, or −1) saturating counter. When the system detects that independent collection is more desirable, it decrements the counter and switches to independent mode if the counter is −1. This avoids oscillation between the two modes when an application has alternating latency-tolerant and latency-sensitive sections. On the other hand, when the system is in independent mode and detects that the application is latency sensitive, it directly sets the counter to 1 and immediately reverts to synchronized mode.

Limitation: We note here a potential limitation of this formulation. It assumes that the number of idle periods is does not significantly change by making each idle time period longer. This is a reasonable approximation for

(1) SPMD-style parallel programs with rare communication, (2) SPMD-style parallel programs without latency hiding, and (3) asynchronous applications with plenty of parallelism in each processor. Our formulation is a reasonable model for (1) and (3) because such applications exhibit very small n under whatever communication latency, so the number of idle periods will not differ much. It is also a reasonable model of (2) because, without any latency hiding, the number of idle periods is approximated by the number of request messages, which is not affected by communication latency in SPMD applications. Important applications that we do not know can be modeled with our formulation are "moderately latency-tolerant" programs where each processor tries to mask latency by techniques such as prefetching and producer-initiated communication or by a moderate number of threads on each processor. If the latency tolerance is large enough to completely mask the latency in normal circumstances, but not enough to hide an entire local collection latency, the system observes a very small n in synchronized mode and might wrongly switch to independent mode, even though the application does not actually tolerate it. Note that this happens only when latencies in synchronous mode are almost *completely* hidden so that each processor exhibits no idle periods for almost all the remote communications. On the other hand, our formulation works correctly as long as latency hiding is typically partial, *i.e.*, the application still observes approximately the same number of idle periods even with latency hiding. They should be examined in more detail for further clarification.

3.5 Experimental Conditions

This section briefly describes characteristics of applications relevant for the following experiments. They are summarized in Table 3.1.

In BH, we build a tree which represents an entire simulation space (BH-tree), traverses the entire tree once, and then traverses part of the tree many times to calculate force for each particle. Live data at a global collection mainly consists of the entire BH-tree, particles, reply channels, and activation frames. The force calculation phase dominates computation time and determines the overall behavior of the application. A communication occurs when a processor accesses a tree node whose copy is not present in the local processor. Each processor sequentially processes particles. Hence, a remote

Application	Main Data	Parallelism	Communication
BH	BH-tree nodes, particles, frames, and channels	SPMD	frequent, synchronous
CKY	CKY matrix, parse trees, frames, and channels	communicating threads	frequent, synchronous
RNA	Frames and channels	parallel recursion	infrequent, asynchronous
GA	Workers and genes	SPMD	infrequent, synchronous

Table 3.1: A brief description of parallel applications

access stalls the accessing processor, making this application very sensitive to communication latency.

In CKY, we invoke $1/2n(n+1)$ concurrent threads for parsing a sentence with n words. Since the length of a sentence is between 35 and 45, we create from 1,000 to 2,000 concurrent threads for each sentence. A thread consumes from 0 to $2n$ values produced by other threads and produces one result using these values. We implement a data structure for the procedure-consumer synchronization (*the CKY matrix*) by concurrent objects. Live data at a global collection mainly consists of the entire CKY matrix, parse trees under construction, activation frames, and reply channels. Communication is frequent and synchronous. The amount of parallelism on each processor depends on the number of processors and the input sentence. For example, while it is enough to have 2,000 threads on 16 processors, it is not enough to have 1,000 threads on 256 processors. Thus we cannot easily predict whether or not CKY is latency-tolerant.

In RNA, parallelism is extracted via parallel recursive calls to a tree-search procedure. There are few globally shared objects. Live data at a global collection mainly consists of activation frames and reply channels. A communication occurs when a processor makes a parallel recursive call, a branch of recursive call terminates, and a processor broadcasts an improved solution to other processors for pruning. RNA typically forks 250K concur-

rent threads, all of which can run independently. Thus each processor has plenty of intra-node parallelism.

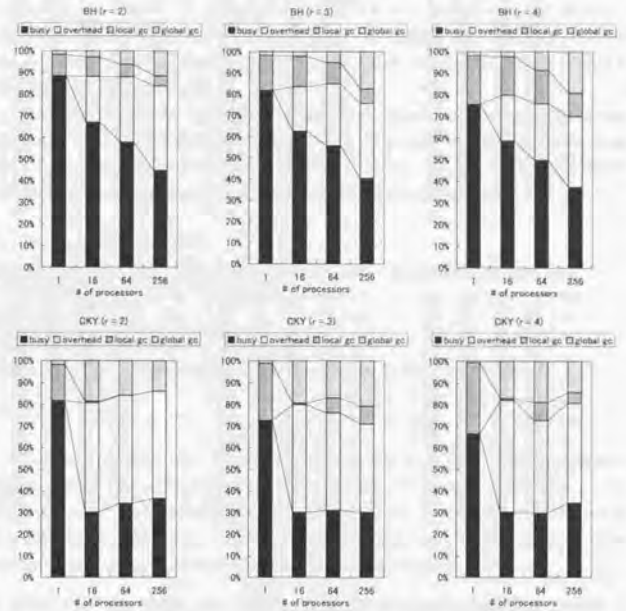
In GA, there is one worker on each processor. Workers independently mutate local genes or crossover two genes that belong to the worker. Live data at a global collection mainly consists of workers and genes. A communication occurs when they exchange their genes. Thus each processor alternates computation-only phases and short communication-intensive phases.

We briefly summarize the performance of the non-GC part of the applications. For BH, CKY, and RNA, we also wrote a sequential version in C++ and compared the sequential performance of ABCL/f with C++. We ran parallel ABCL/f binaries that are executable as parallel applications on workstation clusters. The speed of the ABCL/f versions ranged from about 40% to 55% of that of the C++ versions. These results indicate that ABCL/f has reasonable sequential performance. The speed-up factor for running these applications on 256 processors ranged from 20 to 160, showing that they are reasonably scalable. To sum up, both the application and the language are fast enough to reveal any significant inefficiency in garbage collectors.

3.6 Collection Overhead

Figure 3.4 breaks down the application time into busy, parallelization overhead, local GC, and global GC. Times are totaled over all the processors. The parallelization overhead includes communication overhead and context switch overhead. The breakdown does *not* include idle time, because it never involves allocation requests. By excluding idle time, we effectively estimate an upper bound of the relative overhead of garbage collections. We tested various values for the threshold parameter r that determines whether we collect garbage or expand the heap when an allocation request cannot be served. For each application, we set r to 2, 3, and 4 and ran the application on AP1000+ using 16, 64, and 256 processors and on a single processor UltraSparc workstation. In the experiments in this section, we turned off the adaptive selection strategy of local collections; we always used synchronized local collections.

Overall, the collection overhead ranges from a few percent up to around one fourth of the total time. More importantly, in all applications except



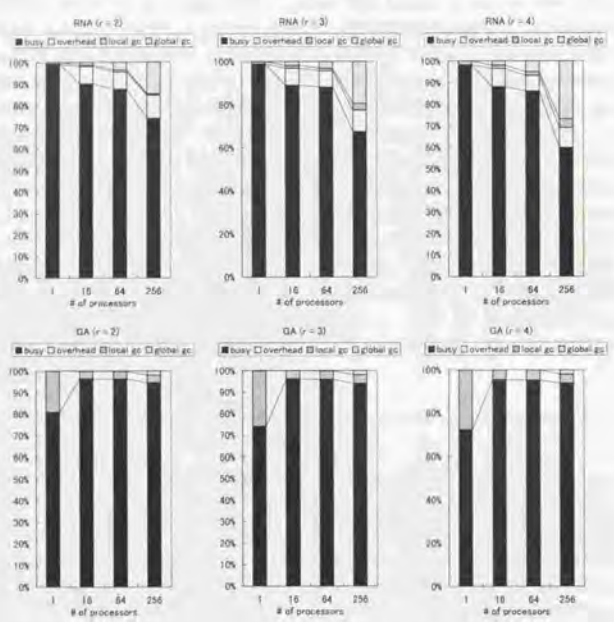


Figure 3.4: Breakdowns of application time into busy, overhead, local GC, and global GC (from bottom to top). Times are totaled over all the processors. Idle times are excluded. The overhead refers to communication and context switch overhead.

RNA, the time spent on collections is essentially constant regardless of the number of processors. Most cases that exhibit large collection times on AP1000+ (BH and CKY when $r = 3$ or 4) also have correspondingly large collection times on the single processor. This indicates that the garbage collection in these applications was at least as scalable as the application itself. Only in RNA did the collection time significantly increase with the number of processors. It turned out that RNA was highly scalable, exhibiting 160 times speed-up on 256 processors. The result merely indicates that our collector is *less scalable* than RNA; it is not noticeably worse in RNA than in other applications.

Table 3.2 shows the number of local/global collections and average pause time of a local/global collection in each application. We only present data for $r = 2$. Numbers in other settings are similar. Figure 3.5 breaks down the overhead of global collections into the following five parts.

Sync: for the sync phase.

Undelivered: for the undelivered phase.

Local: for local marking

Remote: for scanning the exit table and the remote marking.

Idle: Idle time.

For each application, we show the case for $r = 2$ on 256 processors. We see there is no phase that dominates the total collection time in all applications. We also note that the overhead of the undelivered phase is not significant, justifying our design decision as to how to find references in undelivered messages, which was described in Section 3.3.3

3.7 Impact of the Local Collection Strategies

This section examines the impact of the two local collection strategies. Figure 3.6 shows performance of the following four scheduling strategies:

Fixed-Synchronized (FS): Always synchronize.

Fixed-Independent (FI): Never synchronize.

App.	No. of processors	No. of collections (local/global)	pause time (local/global)
BH	16	33/6	253/395
	64	12/5	122/365
	256	4/3	141/461
CKY	16	0/31	-/904
	64	0/5	-/994
	256	0/1	-/1059
RNA	16	19/21	80/95
	64	10/15	55/94
	256	2/15	60/115
GA	16	77/5	99/74
	64	38/3	56/60
	256	14/2	61/100

Table 3.2: Number of local/global collections and their average pause times.

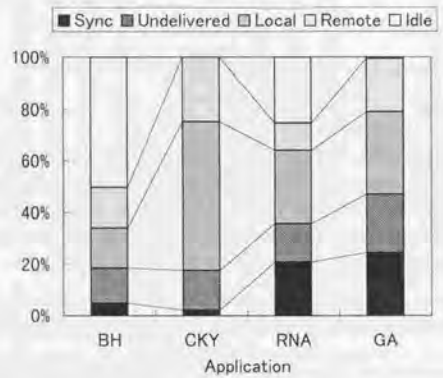


Figure 3.5: Breakdown of the overhead of global collections into sync, undelivered, local, remote, and idle (from bottom to top).

Adaptive-Synchronized (AS): Adaptive, with the initial strategy being synchronized.

Adaptive-Independent (AI): Adaptive, with the initial strategy being independent.

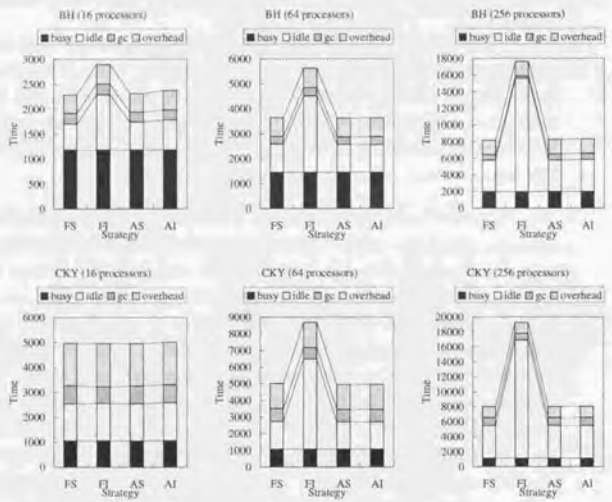
Adaptive strategies choose an appropriate strategy based on the criteria described in Section 3.4.4. The graphs break down the application time into busy, idle, GC (both local and global), and parallelization overhead. Times are totaled over all the processors.

When we compare the two fixed scheduling strategies, neither is consistently better than the other, but we can make several useful observations.

- BH on any number of processors and CKY on 64 and 256 processors significantly suffer from independent collections. More interestingly, the collection time *per se* does not increase at all. It is the *idle* time that increases significantly. That is, a processor that is locally collecting becomes unresponsive to requests from other processors, causing idle time on those requesting processors.

- When the fixed independent strategy is better than the fixed-synchronized strategy, gains are small. Although the number of applications we tested was too small to conclude that this usually holds, we can conjecture that it holds for a wide range of applications based on the following discussion. When allocation rates of processors are fairly well balanced, the synchronized strategy merely triggers collections slightly earlier than necessary on each processor. It only adds a little extra work on the collector. A relatively large penalty is added to the master, but the overhead is still one broadcast. When allocation rates are very unbalanced, on the other hand, non-intensive processors must perform marking which would not be necessary at all. This typically occurs in the initialization phase of a program. Fortunately, in such circumstances, the critical path of the application typically exists on intensive processors and adding extra work on less-intensive processors will not affect overall performance.

- In all cases, adaptive strategies are better than the worst fixed strategy, regardless of the initial strategy and close to the best fixed strategy.



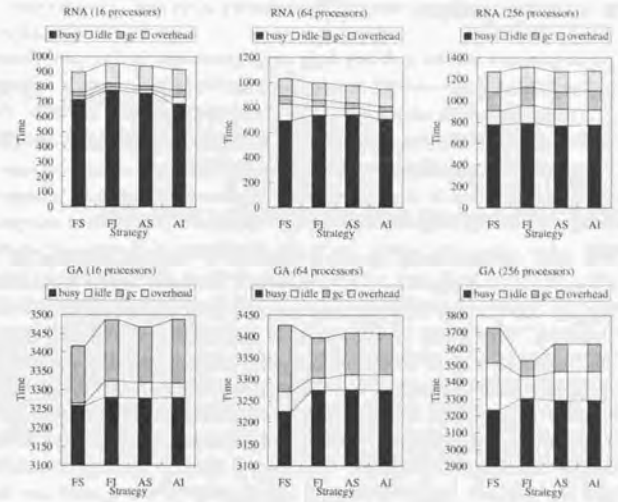


Figure 3.6: Impact of collection scheduling strategies. The graphs break down the application time into busy, idle, GC (local + global), and overhead. Times are totaled over all the processors. *FS* refers to the fixed-synchronized strategy in which we always synchronize local collections and *FI* the fixed-independent strategy in which we never synchronize local collections. *AS* and *AI* refer to adaptive strategies whose initial strategies are synchronized and independent, respectively.

From application logs, we confirmed that adaptive strategies successfully select the right strategy when one fixed strategy is clearly better than the other.

3.8 Discussion

An important conclusion drawn from our experiments is that performance of frequently communicating synchronous applications is heavily damaged by a scheduling skew introduced by independent collections. However, synchronized local collection is not the sole strategy for fighting this problem. Here we discuss alternatives and potential problems.

3.8.1 Incremental/Interruptible Local Collection

The most straightforward approach would be to make an independent local collector interruptible. An independent collection would periodically poll the network and schedule incoming messages, even in the middle of a collection. This is just an adaptation of incremental collection techniques [39]. Expenses include additional memory overhead, polling overhead, and implementation complexity. The viability of this approach will depend on memory requirements of the application. If each processor has plenty of available memory, additional memory overhead caused by incremental collection will cause no problems. However, if memory shortage is detected and the collector wishes to restrain the progress of the application, that processor becomes unresponsive. After all, the system must still have a synchronized collection as the last resort in cases many processors exhibit memory shortage, just as single processor incremental collectors must have full collections in case where memory is so constrained.

3.8.2 Latency-Tolerant Algorithms

As we have seen in Section 3.7, programs in which communication is infrequent or very latency tolerant do not suffer from a scheduling skew. It might be possible for a compiler of a programming language to automatically generate latency tolerant code or at least encourage latency-tolerant programming styles. However, latency-tolerance is achieved at the expense of additional memory requirements and additional scheduling overhead. In

general, it is not a feasible idea to force a programming style in which the programmer must create otherwise useless parallelism just in case the receiver processor is performing a local collection. In order to make sure that the latency is masked when it is very unpredictable, the programmer must overestimate it, leading to excess parallelism and poor utilization of the local storage in usual cases.

3.9 Summary

The suitability of collection schemes on large-scale parallel machines has not been studied enough and has often been misunderstood. In particular, the expense of global synchronization and the benefit of independent local collections have been overstated. Performance of complex systems like garbage collections should be empirically examined, taking their space requirements and interaction to the application into account. Our experiments have shown that the independent local collection is a dangerous strategy that severely slows synchronous applications, by up to 60% in our experiments (CKY on 256 processors). The synchronized local collection exhibits much more robust performance characteristics, despite the cost of global synchronization and the extra work imposed on collectors. With simple techniques which reduce the overhead for message passing and scanning exit tables, the cost of global marking becomes insignificant. In a heap expansion policy which is the most space-intensive in our experiments ($r = 2$), but still not as intensive as collectors used in heap-intensive languages, garbage collection occupies at most 15% of the application time (excluding idle time). Our results indicate that an efficient global collection can be implemented by a simple global marking together with a careful collection scheduling strategy, at least in dedicated parallel computers. Our hope is that this work outlines a 'baseline' implementation strategy of garbage collectors on distributed-memory parallel machines, from which more efficient collectors are derived in the future, under a right definition of "efficiency" and a right framework for performance evaluation.

Chapter 4

ABCL/f—The Language Design

This chapter describes design of ABCL/f, a concurrent object-oriented language. Implementation is outlined through giving mappings from ABCL/f constructs to runtime primitives introduced in Chapter 2 and 3. ABCL/f supports *future* as the means to creating parallelism, first-class channels as the means to synchronization, and concurrent-objects as location-transparent mutable data structures accesses to which are automatically protected.

The rest of this chapter is organized as follows. After giving a brief design overview of ABCL/f in Section 4.1, we introduce basic concurrency primitives of ABCL/f in Section 4.2. Section 4.3 and 4.4 devote to the two data type definition constructs in ABCL/f, namely, concurrent objects and immutable data. Section 4.5 shows program examples that some previous concurrent object-oriented languages have difficulty with. Section 4.6 compares the design of ABCL/f with other language designs.

4.1 Overview

Syntax and sequential constructs of ABCL/f are borrowed from Common Lisp [69]. Unlike Common Lisp, ABCL/f has a simple static type system and enforces type declarations for procedure/method parameters. The current implementation of ABCL/f lacks parametric polymorphism and inheritance. Types for local variables are normally inferred but monomorphic

type declarations are necessary where types are otherwise inferred as polymorphic.[1] To summarize, the type system of ABCL/f is similar to that of Pascal and certainly much less powerful than today's modern languages. We currently side step implementation issues that come from powerful type systems. By concurrent *object-oriented* languages, we simply mean languages which support and encourage concurrent objects—mutable data structure accesses to which are automatically protected.[2]

ABCL/f can be most concisely understood as a concurrent and object-oriented extension to simple statically typed procedural languages. The following is the summary of key extensions key extensions.

Channels: As the fundamental primitive for synchronization, it provides first-class *channels*. A channel is a data structure on which synchronizing read/write can be performed. Channels can be passed to other processes or stored in any data structure.

Future: As the fundamental construct for creating parallelism, it introduces a variant of the *future* construct originally proposed by Halstead [33]. The result value of a future expression is a channel, which we call *reply channel* of the future expression, via which the result of the invocation can be extracted.

Explicit Reply: The reply channel of an invocation is visible from the invoked process and subject to any first-class manipulation. This feature allows us to construct many flexible communication/synchronization patterns in a natural way. For example, by an explicit reply channel, multiple invocations can share a single reply channel, or an invocation can delegate the reply channel to another invocation.

Concurrent Objects: Concurrent objects are supported as a convenient and recommended way for sharing mutable data structures among concurrent processes. A concurrent object is a data structure where a method invocation can roughly be regarded as an *instantaneous* transaction on that object, in the sense that methods never observe intermediate state of other transactions.

[1] For example, `(let* ((r '())) ...)` requires a monomorphic type declaration for `r`, since the type of `r` is otherwise inferred as $\forall\alpha$ `list` α.

[2] Such languages are sometimes termed as concurrent object-based languages.

Concurrent Accesses: While achieving the instantaneousness of a method invocation, we still allow a certain amount of concurrency between multiple method invocations on a single concurrent object. In particular, we guarantee that read-only methods are never blocked by other methods.

4.2 Parallelism and Synchronization Primitives

4.2.1 Channels

Channels are the fundamental entities that realize synchronization and communication between processes. Channels can be explicitly created via the following form:

`(make-channel` *type*`)`,

though they are most often *implicitly* created as the result of a procedure/method invocation as will be described in Section 4.2.2. *Type* denotes the type of values that are stored in the channel. The type of a channel that accepts values of *type* is `(future` *type*`)`.

When c is a channel, we can perform following operations on c:

- `(touch` c`)`—extracts a value from c. The extracted value is supplied to the enclosing expression. If there are no values in the channel, the evaluation of the enclosing expression is blocked until the value is supplied by reply.
- `(reply` x c`)`—puts x in c. The enclosing expression immediately gets a unit.[3] If there is suspended touch operations, the value is feed to one of these touch operations.

Again, these operations can be explicitly used at any place, but are most often called implicitly. As we will see in Section 4.2.2, `reply` and `touch` are implicitly used for communicating the result value of an invocation between the caller and the callee.

There may be multiple values stored in the channel when a touch occurs. In that case it may get any one from the stored values. Similarly, there may be multiple suspended touches when a reply occurs. In that case, it may resume any suspended touch from them.

[3] A special constant typically used when the value returned does not matter.

4.2.2 Procedure Invocation

In ABCL/f, procedures or methods are called either asynchronously or synchronously, and either locally or remotely. In addition, it adds a further flexibility in the way the caller and the callee communicate the result value. In distributed memory machines, a remote procedure call normally requires two messages (*i.e.*, request and reply). This sometimes results in unnecessary round-trip communication. For example, consider processor P wishes to create a local copy of a remote object O. This could be done by writing a method that creates a replica of the receiver object (self) on P and invoking the method from P. This involves an extra round-trip communication because the created replica is first returned back to O, which is then forwarded to the original requester. Instead, the replica created at P should be directly returned to the original requester. We address this kind of issues by allowing the programmer to specify the location to which an invocation should return the result. The way we view a procedure invocation is as follows:

- Any procedure or method takes, in addition to regular parameters, another parameter called *reply channel* via which the caller and the callee can communicate result values.
- The caller creates a new (unique) channel and passes it to the caller as the reply channel unless otherwise specified. The caller can specify any channel as the reply channel when desired.
- Any procedure invocation creates a new thread of control. Whether or not an invocation is synchronous call is a matter of *when* the caller happens to require the result.

Suppose f is a procedure or a method defined by `defun` or `defmethod` constructs described below. The canonical form of a procedure invocation is written as follows:

`(future (`$f\ a_0\ a_1 \cdots a_{n-1}$`) :reply-to` r `:on` o`)`

This creates a thread which evaluates the body of f on processor o and passes $a_0, a_1, \cdots$, and a_{n-1} as arguments and r as the reply channel of the invocation. The value of this expression is r. From this canonical form, shorter and more frequently used forms are derived.

- If keyword `:on` is omitted, f is evaluated on the local processor.
- If `:reply-to` r is omitted, a new channel is created and supplied as the reply channel. That is,

  (`future` ($f\ a_0\ a_1\ \cdots\ a_{n-1}$) `:on` o)
  ≡ (`future` ($f\ a_0\ a_1\ \cdots\ a_{n-1}$)
  `:reply-to` (`make-channel` *type*) `:on` o),

  where *type* is the type of the reply value of f.
- A synchronous invocation is done by immediately touching the result of the future. This is written by now expression:

  (`now` ($f\ a_0\ a_1\ \cdots\ a_{n-1}$) `:reply-to` r `:on` o),

  which is actually an abbreviation of:

  (`touch` (`future` ($f\ a_0\ a_1\ \cdots\ a_{n-1}$) `:reply-to` r `:on` o)).
- Finally, when neither `:reply-to` nor `:on` are specified, now expression can be simply written:

  ($f\ a_0\ a_1\ \cdots\ a_{n-1}$)

  which is the most frequently used form of procedure/method invocations.

4.2.3 Procedures

A procedure in ABCL/f is defined by a toplevel form called `defun`. Its syntax reflects the way in which we view a procedure invocation described in the previous section. The canonical syntax of `defun` is:

(`defun` *name* ($p_0\ p_1\ \cdots\ p_{n-1}$) `:reply-to` r
  (`declare` $\cdots$) *;; type declaration*
  body),

where $p_0, p_1, \cdots, p_{n-1}$ refer to arguments and r to the reply channel of the invocation. Unlike Common Lisp, declare clause is mandatory in ABCL/f and has the following syntax:

declare-clause	::=	(`declare` *type declare*$*$)
type declare	::=	(*type-expression* {*variable-name*}$*$)
	|	(`reply-type` *type-expression*)

Here, (*type-expression* {*variable-name*}$*$) declares listed variables to be of type *type expression*, whereas (`reply-type` *type-expression*) the type of the reply value to be of type *type-expression*.

A `defun` defines a template of threads that, when invoked, execute its *body*. The *body* typically replies a value to r, though neither the compiler nor the runtime system enforces this property. The programmer could write a procedure which reply values multiple times, or do not reply any value at all to r. For example, a procedure may store r somewhere without replying any value and another procedure may obtain the reference to r and reply a value. We later show some examples where this is useful.

The clause `:reply-to` r can be, and in fact often is, omitted. In that case, the definition denotes a template of threads that, when invoked, evaluate *body* and reply the evaluated value to the reply channel. That is,

```
(defun name (p0 p1 ··· pn-1)
  (declare ···) ;; type declaration
  body)

≡

(defun name (p0 p1 ··· pn-1) :reply-to r
  (declare ···) ;; type declaration
  (reply body r)
```

For example, the following code defines a simple procedure which computes the nth Fibonacci number, which takes an integer (`fixnum`) as the parameter and returns an integer.

```
(defun fib (n)
  (declare (fixnum n) (reply-type fixnum))
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

As this example indicates, `defun` in ABCL/f is syntactically similar to the defun in Common Lisp. Differences are it has a declare-clause and optional `:reply-to` clause, and a declare-clause has a reply-type declaration.

Implications to Implementation: The semantics of a procedure call in ABCL/f is strictly based on the view that each procedure invocation has an independent thread of control. There is no inherent notion of "sequential" call; it is just a particular combination of the behavior of the caller and the callee. Even if the caller invokes a procedure by now expression, the callee may reply a result before its termination and continue. In that case the rest of the callee and the caller are semantically parallel. This implies that an implementation of ABCL/f cannot serialize a given invocation solely by looking at its call site; it must consult the definition of the called procedure as well. For example, the call to f in the following code is apparently sequential:

```
(progn
  (acquire-lock x)
  (f x)
  (release-lock x))[4],
```

but if the definition of f was:

```
(defun f (x) :reply-to r
  (declare ( ... ))
  (reply 10 r)
  (acquire-lock x)
      ...
                  ),
```

the compiler cannot serialize the call to f. Under the correct semantics, the implicit touch which occurs at `(f x)` is resumed when `(reply 10 r)` is done. Then the caller performs `(release-lock x)`, making `(acquire-lock x)` in f successful. If the compiler would (wrongly) serialize the call to f,

[4] `progn` executes its constituent sequentially. `acquire-lock` and `release-lock` are hypothetical mutual exclusion constructs which lock and unlock the given datum, respectively. This is actually a lower-level representation of a method that updates an object in ABCL/f.

`(acquire-lock x)` in f would never succeed, resulting in a deadlock which should not occur. We will fully describe our implementation of procedure calls in Section 5.3.

4.3 Concurrent Objects

In ABCL/f, a concurrent object plays two roles. First, it serves as a means to sharing data in a location transparent fashion. A method invocation automatically locates the receiver object and emits a message when the object is remote. Second, it serves as a safe and a stylized means to sharing mutable (updatable) data structure among concurrent threads. The programmer can assume concurrent accesses to an object interleave at the granularity of a method invocation, rather than individual load and stores, without explicitly locking/unlocking objects.

4.3.1 Classes and Methods

Defining Classes

A class is defined by `defclass` and a method by `defmethod` or `defmethod!`. For example,

```
(defclass point ()
  (real x)
  (real y))
```

defines class called point, each instance of which has slots called `x` and `y`. What follows after the class name is the list of inherited classes, which is not yet supported in the current implementation and thus is always empty.

A defclass implicitly defines a function with the class name that creates an instance of the class. For example, an instance of point class is created by:

```
(point 2.0 3.0)
```

Defining Methods

The following defines a method that returns the distance between the point and the origin.

```
(defmethod point distance ()
  (declare (reply-type real))
  (sqrt (+ (* x x) (* y y))))
```

This can be, as usual, called by:

```
(distance p)
```

where p is an instance of point. Note that the first argument of a method invocation specifies the receiver object, which does not appear in the parameter list of a method definition. It is implicit and can be referred to by self in the body of a method.

Unlike regular procedures, an invocation of a method cannot specify : on clause and is always performed on the owner processor[5] of the receiver object. In all other aspects, a method invocation shares the same model as a regular procedure invocation described in Section 4.2.2; it can be called either synchronously or asynchronously and explicit reply channels can be used in methods as well. For example, method distance could also be written by:

```
(defmethod point distance () :reply-to r
  (declare (real dx dy) (reply-type unit))
  (reply (sqrt (+ (* x x) (* y y))) r)),
```

though this is just a clumsy coding style of the previous simpler definition.

4.3.2 Updating States

Updating the state of an object is not expressed by an individual update to instance variables. Instead, it is expressed by become construct, which specifies new values for updated slots and atomically update all the specified variables. To our knowledge, this idea is first described by Yariv in Sympal [7]. For example, the following method increments x and y by dx and dy respectively.

```
(defmethod! point move! (dx dy)
  (declare (real dx dy) (reply-type unit))
  (become (redraw! self) :x (+ x dx) :y (+ y dy)))
```

[5]The current implementation of ABCL/f never performs software caching. The owner processor of an object always refers to the processor that created the object.

The first argument of a become (`(redraw! self)` in this case) is called *result expression* of the become and specifies which value the become is evaluated to, whereas the rest part the updated values for slots. Values for unchanged slots can be simply omitted. A become expression first evaluates all the new values for updated slots, update the slots atomically, and then evaluate the result expression.

Notice that we used `defmethod!` above, rather than just defmethod. The rule is that a become cannot appear in the body of defmethod. We put a further restriction on the position of become inside the body of a `defmethod!`, so that become is performed *exactly once* in a method invocation. In general, this cannot be precisely verified at compile time, hence some syntactic rules that conservatively reject suspicious programs are necessary. Rules must be simple so that they can be told to the programmer. Unfortunately, the syntax of ABCL/f is borrowed from Common Lisp and is not structured enough to express restrictions in a sufficiently simple manner. Specifically, it has an unstructured goto statement, in the presence of which there does not seem to be sufficiently simple and precise rules that describe the restriction. At present, the implementation resorts to runtime checks and performance evaluation turned off the runtime check, assuming that we have more structured constructs in which we can define the restrictions syntactically.

4.3.3 Concurrency and Consistency

ABCL/f object model allows certain amount of concurrency between methods operating on a single object, while preserving a simple way of reasoning about state of an object. Let us call a section between an invocation of an update method and its become *an update section*. Simply stated, ABCL/f object model serializes all update sections on a single object. All other accesses to a single object (either by a non-update section of an update method or by read-only methods) may overlap with other accesses (including update sections). For example, it is safe to invoke a read-only method and waits for its completion from within another update method and vice versa. It is also safe to invoke an update method from within a non-update section of another update method.

How should the programmer reason about state of an object, when

method executions may be interleaved in such a way? First, we guarantee that a become updates all the specified instance variables atomically. That is, a method never observes a combination of slot values that are partially updated. Second, once a method is invoked, the body of the method consistently observes the same values of instance variables, even if slot values are changed by other methods. Intuitively, a method atomically copies all the instance variables at the beginning and operates on the copy, and a become writes back the new slot values atomically.

A special care must be taken when a read-only method (M) calls an update method ($M!$) on self and uses instance variables after the completion of M. M does *not* observe state updated by $M!$ directly from M. M can observe updated state by making another method invocation to self from within it.

Except for this tricky part, ABCL/f object model is simple and intuitive; it provides a model in which all invocations on a single object appear to be serialized and the serialization *respects* the order implied by synchronizations in the program in the following sense. (1) Let M and M' be update methods invoked on an object. If either invocations or becomes of M and M' are ordered in the program, the serialization preserves the corresponding order between M and M'. If the order implied by the invocations contradict the order implied by becomes, the program is unsafe (results in deadlock). A salient example is an update method called from within an update section of another update method. (2) Let M be an update method and M' be a read-only method invoked on an object. If the become by M and the invocation of M' are ordered in the program, the serialization preserves the corresponding order in the serialization.

Notice that when an update method M calls a read-only method M' the invocation of M' proceeds the become by M in the program, thus M' proceeds M in the serialization, despite that M' is called from within M.

4.4 Immutable Data

ABCL/f has immutable data and distinguishes them from concurrent objects by its syntax and static type system. An immutable data type is defined by deftype construct, which is analogue of the datatype construct in ML. An immutable datum may have a reference to a concurrent object,

thus it can be a part of large mutable data structure. For example,

```
(deftype complex ()
  (rectangular real real)
  (polar real real))
```

defines a complex number. This defines a new data type called complex, two procedures (*constructors*) called rectangular and polar, each of which creates a new complex data from two real numbers. For example, the type of both

```
(rectangular 3.0 4.0)
```

and

```
(polar 5.0 (/ *pi* 3))
```

is complex. To access fields of a datum, ABCL/f provides a pattern match expression. For example, the following defines the absolute value of a given complex number.

```
(defun complex-abs (z)
  (declare (complex z) (reply-type real))
  (match z
    ((rectangular x y)
      (sqrt (+ (* x x) (* y y))))
    ((polar r _)
     r)))
```

The functionality provided by deftype can be in theory subsumed by classes with inheritance. Hence, deftype might be somewhat redundant from the language designer's point of view. We incorporated a special construct for defining immutable data because in distributed-memory parallel programs, it is often desirable for the programmer to pass a linked data structure by structure copy, rather than by a reference. Copying a potentially mutable data on distributed-memory machines implies that some coherence protocols must be implemented by software. At present, we sidestep this problem by distinguishing mutable and immutable data. Immutable data are structurally copied on remote communication, while mutable data (concurrent objects) are simply passed by reference.

4.5 Examples

4.5.1 Concurrent Tree Updating

This example demonstrates how the concurrency semantics of our model, the notion of before/after-stage in particular, allows natural description of a concurrent data structure. Consider a binary tree search algorithm where each node of the binary tree is a concurrent object. The example is a model of the tree construction method in Barnes-Hut N-body algorithm. Here is the definition of each node object.

```
(defclass bintree-node ()
  ;; this node associates mapping
  ;; between KEY ↔ VALUE
  (fixnum key)
  (fixnum value)
  ;; children is void when it does not exist
  (bintree-node left)
  (bintree-node right))
```

Each node has its key and associated value. It holds that the key of the left child is less than that of self and the key of the right child is greater than that of self. Hence binary search operation is very straightforward.

```
;;;
;;; Lookup the value associated for K.
;;; return -1 if not found
;;;

(defmethod bintree-node lookup (k)
  (declare (fixnum k) (reply-type fixnum))
  (cond ((= k key) value) ; found
        ((< k key)
         ;; look for the left subtree if K < KEY
         (if (voidp left) -1 (lookup left k)))
        (true
         ;; look for the right subtree if K > KEY
         (if (voidp right) -1 (lookup right k)))))
```


Since this operation does not update the tree, we use `define-method`, hence multiple lookup invocations can simultaneously operate on a single tree. The following method associates element val with key k.

```
;;;
;;; Establish association K ↔ VAL maintaining the
;;; following invariant:
;;;    "KEY of LEFT < KEY of SELF < KEY of RIGHT"
;;;

(defmethod! bintree-node insert! (k val)
  (declare (fixnum k val) (reply-type unit))
  (cond ((< k key)
         (if left
             ;; if there is already left child delegate this value
             ;; to the child unlocking self
             (become (insert! left k val))
             ;; if there is no left child create it
             (become unit
                :left (make-leaf-bintree-node k val)))
        ((= k key)
         ;; an object is already installed in the same key do nothing
         (become unit))
        (true
         ;; the same algorithm as the first case but for the right child
         (if right
             (become (insert! right k val))
             (become unit
                :right (make-leaf-bintree-node k val)))))))
```

This method first finds the appropriate place to which we insert the item and then installs a new node to the place. An interesting case happens in internal nodes; an internal node recursively calls insert! method for an appropriate child *after* it unlocks self for subsequent requests. This is expressed by:

```
(become (insert! left k val))
```

at line 6 and

```
(become (insert! right k val))
```

at line 15. As has been described in Section 4.3.3, these recursive calls are done after the object has been updated, hence do not result in deadlock.

4.5.2 Synchronizing Objects

To demonstrate the expressive power of explicit reply channels, consider an implementation of an object that embodies an application-specific synchronization constraint. That is, upon a method invocation, the object may not be ready for executing the method and wish to defer the execution of the method until certain synchronization constraints are satisfied. Since the synchronization constraints may be satisfied only by subsequent methods the same object, the method cannot simply block computation *inside* the method. We wish to have a way to terminate the current method without replying any answer. This situation actually arose in our implementation of CKY algorithm [54] for parsing context free grammars.

For a simple example, consider implementing a "barrier synchronization" object. A set of processes shares a barrier object and each process invokes finished! method on the barrier object when its local computation has been done. Finished! method does not reply any acknowledgement to the process *until all the processes invoke a* `finished!` *method.*

Here is the definition of barrier class.

```
(defclass barrier ()
  ;; number of finished to wait
  (fixnum n)
  ;; number of finished so far processed
  (fixnum count)
  ;; list of reply channels
  ((list (future unit))  waiters))
```

An instance of a barrier class has three instance variables `n`, `count`, and `waiters` where `n` is the number of `finished!` calls to be synchronized, count the number of `finished!` which have been made, and waiters the list of reply channels of previous calls. When synchronization is realized, that is, nth call to this object is made, it replies value unit to all the channels in waiters as well as the current reply channel.

Method `finished!` facilitates explicit reply channel for deferring the replies.

```
;;;
;;; When this finished is the last call it unblocks
;;; all the waiters by explicitly calling reply otherwise
;;; it does not reply anything so that the caller is blocked.
;;;

(define-method! barrier finished! () :reply-to r
  (declare (reply-type unit))
  (if (= (+ count 1) n)
      ;; reply unit to every channels
      (become (dolist (x (cons r waiters)) (reply unit x)))
      ;; reply nothing
      (become unit :n n :count (+ count 1)
         :waiters (cons r waiters))))
```

In the above, the reply channel is named r. If `(+ count 1)` $<$ `n` (*i.e.,* this is not the last invocation), the method stores r in list `waiters`, replying nothing to r. In the last invocation, the method broadcasts a reply for every reply channel so far received.

4.6 Comparison to Other Language Designs

4.6.1 Concurrent Object-Oriented Languages

A *concurrent object* refers to data that embodies some access arbitration mechanisms so that an execution of a method never observes inconsistent state of an object. Several object models have been proposed and they differ in the degree of concurrency on a single object, therefore the range of deadlock free programs.

Actors and Early Concurrent Object-Oriented Languages

The original Actor model [2] and some early concurrent object-oriented languages such as ABCL/1 [79, 80] and Cantor [8] achieves the instantaneousness of a method execution by mutually excluding all the method invocations

on an object. This is often explained by "an autonomous object that has its own thread and message queue." Although the traditional Actor model gives us the instantaneousness and a very simple model in which the programmer reasons about deadlock, it is often criticized to serialize too much. This not only loses performance gain that is otherwise possible by exploiting parallelism, but also enforces unnatural description of algorithms to solely avoid potential deadlock.

Concurrent Aggregates

Concurrent Aggregates (CA) [19] supports *aggregates* in addition to regular objects. A regular object is a traditional Actor and an aggregate is internally composed of multiple objects, but externally viewed as if it were a single object. By processing multiple method invocations on an aggregate by multiple internal objects, an aggregate can serve as a non-serializing object. Maintaining the consistency among multiple internal objects, if required, is the responsibility of the programmer.

UFO and Sympal

An object in more recent languages such as UFO [62, 63] and Sympal [7] allows/guarantees more parallelism than the traditional Actor. ABCL/f also belongs to this category and UFO, Sympal, and ABCL/f are common in many ways. First they support multiple paradigms, in the sense that they do not force programmers to use concurrent objects wherever concurrency is required. This avoids serializing computation that does not require shared mutable data. Second, a method in those languages allows subsequent methods on an object to overlap with the current method after the current method reaches a certain point. In UFO, the compiler statically identifies a point after which instance variables are never updated and unlocks the object when the execution reaches that point. Become construct in ABCL/f was first proposed by Yariv in the language Sympal, under the name finally [7].

C++ Dialects

Here we only discuss C++ dialects that support object-wise concurrency control mechanisms and do not discuss a notable data-parallel extension

pC++ [13, 14].

CC++ [17] does not directly support concurrent objects, but the similar effect can be achieved by atomic member functions. By declaring a member function as atomic, the member function locks/unlocks the object at invocation/termination as in the traditional Actors. Thus the object model of CC++ has the same problems with early concurrent object-oriented languages. Non-atomic functions can run concurrently with others, but this merely leaves consistency issues for the programmer.

Objects in ICC++ [20] allow two methods M and M' to operate on a single object in parallel if there are no read/write or write/write conflicts between them *on any instance variable of the object*. The main difference between ICC++ and the UFO, Sympal, and ABCL/f group is that the ICC++ model performs mutual exclusion on per instance variable basis, rather than per object basis.

The range of programs which are guaranteed to be scheduled without deadlock do not seem quite different between ICC++ and ABCL/f. A foreseeable problem with the ICC++ object model is that each object now potentially has to have multiple locks to serialize only conflicting methods. The worst case requires a lock per instance variable and removing redundant locks requires global information on the source code.

4.6.2 Other Parallel Languages

Multilisp

Multilisp [33] is the language that originally embodies the future construct. The central idea of future that a future expression returns something that later becomes the result value is adopted not only in parallel Lisps but also in some concurrent object-oriented languages [36, 78].

ABCL/f also supports a variant of future. An apparent difference between the future in Multilisp and the one in ABCL/f is that in Multilisp, producer-consumer synchronization of a future invocation is implicit in value reference, whereas ABCL/f requires explicit touch operations. For example, invoking `(f x)` and `(g y z)` in parallel and then adds the two results is written in Multilisp as:

```
(+ (future (f x)) (future (g y z))),
```

whereas in ABCL/f it is written as:

```
(let ((l (future (f x)))
      (r (future (g y z))))
  (+ (touch l) (touch r))).
```

Informally, the Multilisp view of a future is that what is immediately returned by a future expression is a placeholder object, which later *becomes* the result value for itself, whereas the Schematic view is that a future expression returns a placeholder *into which the result value is stored*.

There are tradeoffs between the implicit and the explicit version. The implicit version, as the above example indicates, often results in a terse expression but loses some flexibility. By making touch explicit, we can distinguish a reference to the placeholder itself from the reference to the value that is stored in the placeholder by the program text. This not only guarantees fast value reference without additional compiler analysis [67], but also gives us more expressive power by making the placeholder first-class citizens. Examples have been given in Section 4.5.2.

Another difference is their positions on shared mutable data. Multilisp provides Scheme built-in data as the basis for mutable data and some atomic memory operations such as `replace-if-eq` (analogue of *compare & swap*). No higher-level mechanisms for defining safe mutable data are provided. ABCL/f supports and encourages the use of concurrent objects to represent mutable data, concurrent accesses to which are arbitrated by the runtime system.

Concurrent ML

Concurrent ML (CML) [58] extends SML by first-class channels and fork (spawn). The main difference is that channels in CML are very *orthogonal* to the original sequential constructs, whereas channels in ABCL/f are *integrated* into sequential constructs. For example, any procedure or method in ABCL/f are callable both in asynchronously and synchronously, while functions in CML are not.

Consider how to perform two CML function calls `f x` and `g x` in parallel. Since the results must now be extracted from a channel, let us define a 'wrapper' function that takes a channel and sends the result of `f x` to the channel.


```
fun wrapper f x c = send (f x, c)
```

What remains is to create two channels, spawn two wrappers, and wait for the result.

```
let c0 = channel ()
and c1 = channel ()
in
     (spawn (fn () => wrapper f x c0);
      spawn (fn () => wrapper g x c1);
      accept c0; accept c1)
end
```

Presumably, a fragment like this will appear very often and should be more stylized, as in ABCL/f. In fact, a restricted version of future can be defined in CML by

```
fun future f x =
  let c = channel ()
    in
      (spawn (fn () => send (c, f x)); c)
    end.
```

Except that it can only invoke a unary function, the above future takes any function and any argument and returns the reply channel. This is more monolithic and less flexible than futures in ABCL/f, in that a future now always creates a reply channel and the caller loses the chance to specify a reply channel.

Given that a function is the fundamental building block of CML programs, CML should support and encourage a convenient way for invoking functions in parallel. ABCL/f is designed based on this principle, while leaving chances to construct customized communication structure whenever desired.

Chapter 5

Implementation of ABCL/f

5.1 Overview

The compiler translates ABCL/f programs into C++ programs, which are then compiled by GNU C++ compiler. The generated code (syntactically) relies on extensions supported by GNU C++ compiler. In particular, it extensively uses *statement expressions*,[1] which are expressions that may contain arbitrary control statements inside. Since the implementation of StackThreads already relies on GNU C++ in much more fundamental ways, we did not hesitate to rely on it also in the code generator.

The compilation from ABCL/f to C++ roughly consists of two phases. The first phase (or, "frontend") transforms constructs that are apparently different but are actually similar into a combination of "essential" constructs. For example, various loop constructs such as do, dolist, and dotimes are translated into a combination of blocks and goto expressions. Toplevel definitions such as defmethod and defun are translated into a canonical procedure-definition construct, which defines an asynchronously invoked procedure. Diverse calling sequences including synchronous call, asynchronous call without an explicit reply channel, and asynchronous call with an explicit reply channel, are converted into a combination of a channel creation, touch, and a canonical sequence in which the reply channel is explicit and the invocation is asynchronous. The two data definition constructs (*i.e.*, `deftype` and `defclass`) are also unified into a canonical type-definition construct that names a data format. The task carried out by the frontend is essentially a simple macro expansion that reduces the number of primitives that must be recognized by later phases.

[1] The syntax of a statement expression is almost equivalent to that of a compound statement in C, except that statements are enclosed by '({' and '})' instead of just braces, and the last statement must be an expression. A statement expression is evaluated to the value of the last expression.

The second phase (or, code generator) takes the expanded form and generates C++ code. A procedure definition (expanded either from defmethod or defun) is converted into a single C++ procedure which takes parameters as well as a reply channel as its parameters. A type definition (expanded either from deftype or defclass) is converted into a C++ typedef statement. An ABCL/f expression is converted into a single C++ statement expression that represents the value of the ABCL/f expression in C++. The generated C++ expression retains roughly the same control structure and variable scopes as the original ABCL/f expressions. That is, the transformation is relatively straightforward, in the sense that it does not introduce many temporary variables or breakdown compound expressions into small sequences. Although not empirically verified, this tends to produce a C++ code that is likely to be successfully optimized by the backend C++ compiler.

This style of relatively simple code generation scheme should not be taken as granted, especially in the context of parallel languages. In fact, parallel programming languages are typically compiled into an assembly or assembly-like C code in which compound statements are converted into sequences of small operations. This is partially because a thread may block execution in the middle of a compound expression, and values that are live across the blocking point must be preserved. If one wishes to implement a compound expression by a corresponding compound expression *within a single C procedure*, there must be a way to restart a computation from the middle of a C procedure. Traditional thread libraries accomplish this by allocating a stack for each thread and by switching stack pointer on blocking, suffering from the large resource requirements and thread creation overhead. Therefore fine-grain parallel languages typically manage context explicitly to bypass the stack frame management mechanism of C. When a blocking occurs, generated C code explicitly saves live values into an explicitly managed frame. However, we cannot precisely determine the set of live values at a point within a compound expression, because it depends on the order in which sub-expressions of a compound expression are evaluated. For

example, suppose we straightforwardly compile a compound expression:

$$A + B$$

into a C expression:

$$A' + B'$$

where A' and B' are generated from A and B, respectively. Notice that the evaluation of A' and B' are not explicitly ordered. We further assume that A may block. If the C compiler evaluates A' before B' and A' blocks, live values at the blocking point include all the values which the evaluation of B' requires. If we evaluate A' after B', on the other hand, live values at the blocking point does not include these values, but instead include the result value of B'. Thus we must explicitly order them as in:

$$\begin{aligned} t &= A'; \\ s &= B'; \\ r &= t + s; \end{aligned}$$

or,

$$\begin{aligned} s &= B'; \\ t &= A'; \\ r &= t + s \end{aligned}$$

Note that we not only must order them, but also name each intermediate result, so that they can be saved at blocking points.

Our code generator emits straightforward C code; when blocking occurs, the generated code calls a C procedure, which saves callee-save registers and the whole stack frame for the calling procedure. No matter how expressions are evaluated, the C compiler preserves necessary information on the stack.

5.2 Procedures

The frontend generates three C++ procedures from a regular procedure (one defined by a defun), and two C++ procedures from a method (one defined by defmethod or defmethod!). It generates *body stub*, and *handler* for a regular procedure, whereas *body* and *handler* for a method. A *stub* is a small C++ procedure called either when non-trivial `:on` clause is supplied or the called procedure is not known. In addition to ABCL/f-level parameters and the

reply channel, it takes a parameter that receives the value specified after the `:on` keyword. It checks if the `:on` parameter refers to the local processor and either calls the body or generates a remote procedure call request, depending on whether the call is local. It returns the reply channel as the result value. The skeleton of a stub is shown below. When `:on`+ clause is not specified and the called procedure is known, the caller directly calls the body. For defmethod, we do not generate a separate stub. The body first checks if the receiver object is local and either generates a remote procedure call request or simply continues the method execution, depending on the location of the object. A *handler* is a small C++ procedure that is invoked when a remote procedure call request arrives at a processor. It extracts arguments and the reply channel from the message and invokes the body. Since we generate a specialized handler for each defun and defmethod, and the garbage collector does not require buffered messages be understandable by the collector, a message does not have to be tagged. A *body* is a C++ procedure that takes ABCL/f-level parameters plus the reply channel as C-level parameters. It returns the reply channel as the return value. Figure 5.1 shows a skeleton of a stub, a handler, and a body of a regular procedure and a method.

The different code placement between regular procedures and methods comes from the typical calling sequence for each type of procedures. In the current implementation, our compiler never optimizes away the locality check for a method invocation, although the receiver is often local at runtime. Thus it is important to optimize the sequence that performs a locality check followed by the execution of the method body. For regular procedures, on the other hand, many calls are *statically* known to be local, thus we wish to avoid comparing `:on` parameter and the local processor number. Note that in either case, we do not duplicate the body.

A simple ad-hoc optimization that has not been implemented is a recognition of *simple* procedures that:

- never block,
- do not explicitly access the reply channel, and
- is small.

For such a procedure, it may be worth generating two versions of its body. One is for the regular calling sequence that takes the reply channel as a

```
/* A skeleton of a stub for a regular procedure It takes an additional
   parameter on */
channel<T> * f (channel<T>* r, a_0, a_1, ··· , a_{n-1}, int on)
{
    if (on == local_PE) {
        f_body (r, a_0, a_1, ··· , a_{n-1});
    } else {
        msg[0] = f_handler;
        push_msg (r, msg);
        push_msg (a_0, msg);
        push_msg (a_1, msg);
            ···
        push_msg (a_{n-1}, msg);
        send_msg (on, msg);
    }
  return r;
}
/* A skeleton of a handler for a regular procedure or a method It
   extracts parameters from the message and executes body */
void f_handler (char * msg)
{
    r = extract_msg ();
    a_0 = extract_msg ();
    a_1 = extract_msg ();
        ···
    a_{n-1} = extract_msg ();
    f_body (r, a_0, a_1, ··· , a_{n-1});
}
/* A skeleton of a body for a regular procedure */
channel<T> * f_body (channel<T>* r, a_0, a_1, ··· , a_{n-1}, int on)
{
   /* body of f. no locality check. */
   ··· /* do whatever */
  return r;
}
/* A skeleton of a body for a method */
channel<T> * f_body (channel<T>* r, self, a_0, ··· , a_{n-1})
{
    if (is_local (self)) {
        /* execute body of f */
    } else {
        msg[0] = f_handler;
        push_msg (a_0, msg);
        push_msg (a_1, msg);
            ···
        push_msg (a_{n-1}, msg);
        send_msg (on, msg);
    }
  return r;
}
```

Figure 5.1: A skeleton of a stub, handler, and a body of a regular procedure and a method.

parameter. The other is a specialized interface for local calls without an explicit reply channel. The specialized interface does not take the reply channel and returns the value of the body. A synchronous call can be performed by a direct procedure call to the specialized body. An asynchronous call is done by first calling the specialized body, and then creating a reply channel that stores the return value. Note that the version for general calling sequences are still necessary for supporting remote calls, explicit reply channels, and first-class procedures.

5.3 Procedure Invocations and Context Switches

The frontend expands any type of calling sequence into the following canonical form:

(future ($f\ a_0\ a_1\ \cdots\ a_{n-1}$) :reply-to r)

where f is either a body of a regular procedure, a stub of a regular procedure, or a body of a method. The code generator simply translates this call into a C++ procedure call. Recall that stubs and bodies return the reply channel as the return value. Thus this correctly transforms an ABCL/f expression into a C++ expression that represents the value of the expression in C++.

A procedure blocks when a touch operation does not find any value in the channel. A touch may be explicit in the source code, or automatically inserted by the frontend to implement a synchronous call or a mutual exclusion for a concurrent object. When a touch fails, the procedure allocates its heap context, enqueues the pointer to the context into the channel, and unwinds the stack by calling `switch_to_parent`.

When a thread later writes a value to the channel, the thread moves the resumed heap context to a global scheduling queue. The global scheduling queue is periodically checked and threads in the global scheduling queue are resumed by `restart_thread`. Alternatively, when a reply finds a thread sleeping on a channel, it could immediately resume the thread by directly calling `restart_thread`, rather than inserting the thread in the global scheduling queue and later picking it up. This approach would eliminate the queue manipulation overhead. Unfortunately, this approach has a bad interaction with the conservative garbage collector. A procedure typically

performs a reply at the end, because typical procedures do not have an explicit reply channel. When this is the case, the context of the current thread should desirably be removed from the stack before pushing the context of the next thread. Otherwise, the context of the current thread, which is actually no longer necessary, is identified as a root by the garbage collector. By inserting the restarting thread in the global scheduling queue, we effectively defer pushing the context of the thread. If the reply is the last or near the last action of the current thread, the context of the thread will soon be removed from the stack.

By the same reason, we free a heap context as soon as the thread is resumed. That is, when a single procedure invocation blocks multiple times, it does not reuse the heap context, though StackThreads itself allows the reuse. When a procedure would reuse the heap context after a resume, all the live values of the procedure at the time it blocked last time would be retained until the context is overwritten by subsequent blocks. We instead simply free the heap context by calling `GC_FREE`[2] explicitly, which nullify the context. Other options that we have not tested include:

- Only nullify the context, without freeing the context. This saves allocation cost and initializations unnecessary for second or later blocks. A disadvantage is that it retains the heap context that would otherwise be reused by other purposes, including blocking other procedure invocations.[3]
- Do nothing. Just let the garbage collector reuse the heap context. This is locally an optimal solution, since this incurs no overhead. Hidden cost is unnecessary heap growth or more frequent garbage collections when we do not have enough memory.

Stated above are all the basic mechanisms we have for implementing procedure calls. This correctly implements all the calling sequences supported by ABCL/f, including synchronous and asynchronous calls, local and remote calls, and calls with or without explicit reply channels. Everything is derived

[2] A procedure supplied by Boehm & Weiser's GC that frees a region of memory allocated by `GC_MALLOC`.

[3] Many heap contexts have more or less a similar size. Hence freed heap contexts have a plenty of chances to be reused soon.

from the canonical calling sequence, touches, and replies. Let us see several typical examples to understand how this mechanism works.

Example 1, Synchronous calls: A synchronous call is just a combination of an asynchronous call + touch immediately after the call. For example, a synchronous call:

```
(f x)
```

is expanded by the frontend into the following canonical form:

```
(touch (future (f x) :reply-to (make-channel type))),
```

where *type* is the reply type of `f`. Suppose this call is local (*i.e.*, either `f` is a regular procedure or `f` is a method and `x` is a local object). A channel is created, `f` is called with the channel and `x` as arguments. Since the call is local, `f` starts its computation. `F` returns to the caller either when `f` terminates or blocks. Either case, the caller then tries to touch the channel. If the value exists, the caller simply proceeds. Otherwise it blocks. Now suppose the call is remote. In this case, `f` emits a remote procedure call message to the remote processor and immediately returns. The caller then tries to touch the channel, finds the channel to be empty, and blocks. Note that the caller does not perform a particular check to see if `f` is blocked, or the call was performed locally. It simply checks if it can proceed.

Example 2, Asynchronous calls: Suppose we have:

```
(let* ((l (future (f x)))
       (r (future (f y))))
  ...
  (+ (touch l) (touch r))),
```

where `f` is a method, and we do not know statically whether objects are local or remote. Let us further assume `f` is a small method that, once started, always reply a value without blocking. First consider the case where both `x` and `y` happen to be local at runtime. In this case, both method invocations schedule `f`, which reply a value to the reply channel before it returns to the caller. When the caller later tries to

touch `l` and `r`, it will find a value from both. Next consider the case where `x` happens to be remote. The invocation `(future (f x))` sends a remote procedure call request to the remote processor, returns to the caller with an empty channel. Then `(future (f y))` is called and the caller further proceeds to '...'. Meanwhile, the reply from `(f x)` may or may not arrive. When it does, `(touch l)` will find a value in `l`. This is a representation of latency hiding in ABCL/f. Only when the reply has not arrived until `(touch l)`, does the caller block.

Example 3, A chain of Synchronous calls: Suppose a chain of procedure invocations $f_0, f_1, \cdots, f_{n-1}$ where f_i calls f_{i+1} synchronously and locally and f_{n-1} blocks. We further assume every procedure replies a value to the reply channel if and only if it terminates. Since f_{n-1} calls f_{n-2} synchronously, blocking f_{n-1} will cause f_{n-2} also block, which will in turn cause f_{n-3} also block, and so forth. In this way, this cascading block continues until f_{n-1} is resumed. When f_{n-1} is resumed, its context is copied on stack (via `restart_thread`) to restart it. It will eventually terminate and resume f_{n-2}, which in turn eventually resume f_{n-3}, and so on. Notice that no particular mechanisms are provided for maintaining the call chain between f_i and f_{i+1}. They are implicitly maintained through sharing the reply channels between them. Also notice that heap contexts are lazily copied back to the stack; when f_{n-1} is resumed, the copied back to the stack is only the frame for f_{n-1}. Other frames still remain in the heap. That is, if f_{n-1} blocks again, only the frame for f_{n-1} must be saved.

5.4 Unboxed Channels and Efficient Communication via Channels

We have seen that how the combination of the canonical calling sequence and channels implement various calling sequences uniformly. What is left unclear is how to implement channels, which are ubiquitously used in the mechanism. A naive implementation would represent a channel as a pointer to a heap-allocated datum that has two queues, one for values and the other for threads sleeping on it. With this naive implementation, however, the above mechanism is just an expensive representation of various calling se-

queues. The goal of this section is to develop a mechanism that implements the semantics of the first class channels correctly, while achieving efficiency where more specialized mechanisms are be applicable.

Omitting details, we apply a special (*unboxed*) representation for channels that satisfy conditions described below and lazily convert them to the normal (*boxed*) representation (*i.e.*, pointer to a heap-allocated datum) when they no longer satisfy the conditions. An unboxed channel represents its entire state in local variables (which will hopefully be allocated on registers) and reply/touch on it simply update the local variables. The essential condition under which this "in-place update" correctly implements the semantics of channels is that the channel is not aliased, or if it is aliased, a protocol correctly propagates the change to other references.

In our protocol, a channel is created in its unboxed form and remains unboxed as long as:

- it is empty or it has only single value stored in it,
- it is not referenced from heap,
- the thread that creates it references it through at most one local variable, and,
- other threads that reference it do so only through the reply channel and have not blocked.

Put differently, a channel is created in its unboxed form, passed to another local thread via the reply channel parameter *as is*. It must be converted to the boxed representation, however, when it is stored into heap data, passed to a remote thread, passed to another thread via a regular parameter, or aliased to multiple local variables within a thread. How to generate code that maintains the invariant is yet unclear and we detail the code generation scheme in Appendix B.

As long as a channel is referenced only from a single thread, this mechanism works with no surprise. Since it is referenced only from a single variable, updating the variable sufficiently updates all the references to the channel. Less obviously, a channel can be shared among multiple local threads, as long as it is passed to these threads via the reply channel parameter and these threads have not blocked. For example, an expression:

```
(let* ((r (future (f x) :reply-to (make-channel fixnum))))
  ...
  (touch r))
```

creates a channel at the future call and shares it between f and the caller through the reply channel parameter. We define a protocol by which an invoked procedure propagates the updated state of the reply channel to its caller. More precisely, an invoked procedure receives a (possibly unboxed) channel via the reply channel parameter, keeps it unboxed as long as the conditions are met, and *notifies the caller of the updated representation* when it returns to the caller (whether by termination or blocking). If it blocks, the channel must be boxed, so that the caller and the callee may share them afterwards. If it terminates, on the other hand, it returns the channel as is, which is hopefully still unboxed.

A channel is represented by a single 32-bit word represented either in boxed form or in unboxed form. A boxed form is simply a pointer to a heap-allocated channel. An unboxed form is either:

- a special value `UNBOXED_EMPTY`, or
- a pair $\langle$*value*, `UNBOXED_ONE_VALUE`$\rangle$

Currently, we assign one to `UNBOXED_EMPTY` and three to `UNBOXED_ONE_VALUE`. Due to the limitation of the word size, we can use only 30 bits for encoding the value stored in a channel. Hence, the current implementation uses unboxed channels only for channels of unit, boolean, character, and fixnum, assuming fixnum is represented in 30 bits. We encode the pair $\langle$*value*, `UNBOXED_ONE_VALUE`$\rangle$ by $(4 \times$ *value* + `UNBOXED_ONE_VALUE`$)$.

It is clearly desirable to use unboxed channels for other data, especially for floating point numbers and pointers, and there are in fact no fundamental difficulties. We do not so simply because of the current implementation artifacts. Since a floating point number fully utilizes 32 or 64 bits, in order to use unboxed channels for floating point numbers, a channel must be split into two words, one for the value and the other for the tag. This can be done by representing them with a C++ struct value or two separate C++ values in the generated code. The former cannot be used because it invalidates the restriction imposed by StackThreads that aggregate data are not allocated on the stack. Thus, we must use the later. However, the current code

generation scheme works by translating a single ABCL/f expression into a *single* C statement expression and there are no ways to represent multiple C++ values by a single C++ expression. Using unboxed values for pointers should be even easier, because two lowest bits of a pointer are in any case zeroes. This is really a silly limitation of the current implementation that uses only a single encoding scheme over all types of channels.

Chapter 6

Application Benchmark

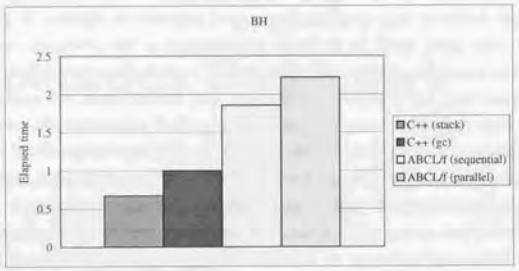
Although results shown in Chapter 2 and 3 demonstrate multithreading and memory management overhead are not significant, we are still interested in how good was the performance of such systems overall, especially relative to efficient sequential systems (such as C and C++). This chapter examines the overall performance of the ABCL/f system by application benchmark. We tested the same applications with Chapter 2 (BH, CKY, and RNA). For each application, we show single processor performance, breakdown of parallel execution overhead, and overall speed-up. Refer to Appendix A for a more though description of each application.

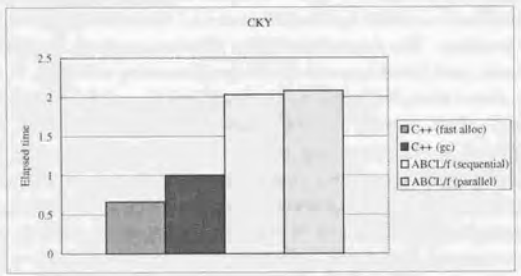
6.1 Single Processor Performance

For each application, we wrote programs both in C++ and ABCL/f, using an essentially the same algorithm and ran them on a single processor workstation (UltraSparc, 167 Mhz with 128 MB memory). The baseline C++ programs are sequential. For ABCL/f, we wrote both parallel and sequential version. The parallel version exploits parallelism when executed on multiprocessor systems. The sequential version does not have overhead for polling and locality checks of concurrent objects. It also eliminates some application specific overheads that are unnecessary on single processors and are easily removable without significantly restructuring the application. The sequential version still incurs, however, overhead for fork, channel creation, communication via channels, and object locking. They are, in general, necessary for implementing semantics of ABCL/f. C++ programs use Boehm & Weiser's conservative garbage collector. ABCL/f and C++ programs exhibit similar heap-allocation behavior, except that some ABCL/f programs allocate many (boxed) channels on heap for synchronization and obtaining results of procedure calls. Using garbage collector for C++ is not meant to underestimate performance of C++ programs. In fact, the conservative garbage collector has an allocation speed superior to malloc. It allocates a small fixed sized block in 11 Sparc instructions in the common case. For allocation intensive applications, the overall performance is much better than programs that use malloc and free on a per datum basis. Of course, it may incur high overhead compared to programs that customize allocation methods, taking advantage of the application specific knowledge about allocation behavior and lifetime distribution of objects. When this is the case, we also write a C++ program with such a customized memory allocator.

Figure 6.1 shows performance of the three applications. Graphs show the relative performance of various versions, with the C++ program that use Boehm & Weiser's GC as the baseline. For each program, "ABCL/f (parallel)" refers to the parallel ABCL/f binary, "ABCL/f (sequential)" the sequential binary, and "C++ (gc)" the baseline C++ program.

For BH, the parallel binary runs about 2.2 times slower than the baseline C++ program. The sequential version removes overhead for polling, locality checks, and (most importantly) software caching overhead. It is still 1.8 times slower than the baseline. It turned out that this high overhead was due to the lack of using unboxed channels for pointer data, as mentioned in Section 5.4. The execution time of BH is dominated by the force calculation, whose main procedure is a recursive method that returns a three dimensional vector, which is represented by a (`deftype`) record in ABCL/f. The representation of the three dimensional vector is a pointer to a heap-allocated record, thus a result of a recursive call is always written in a boxed channel by the callee and then fetched by the caller. The instruction count at leaves of the call tree, which essentially computes a Newtonian force between two particles, is about 65-70 instructions in C++, while it is 90-100 instructions in ABCL/f. The difference comes from the cost for writing the result to the (boxed) reply channel, which is about 25 instructions in the current implementation. The 25 instructions involve ones to make sure that no threads are waiting on the channel, to make sure that the value queue





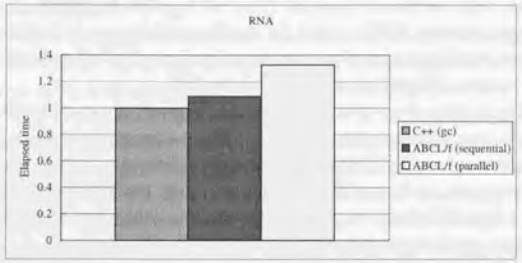


Figure 6.1: Single processor performance of ABCL/f programs, relative to sequential C++ programs. C++ (gc) refers to the baseline C++ program. ABCL/f (sequential) does not incur overhead for object locality checks and polling. It still incurs overhead for fork, communication via channels, and object lock. ABCL/f (parallel) refers to a true parallel binary.

is empty, and to write the value to the channel. Non-leaf nodes of the call tree perform recursive calls to children and each recursive call takes about 25 instructions (at the call site) in C++, while it takes 45 instructions in ABCL/f. The difference again comes from the cost of obtaining the result of recursive calls from a channel, which is about 15 instructions. They include instructions to make sure that only one value is stored in the channel, to read the value, and to make the channel empty. In addition to the difference in the call site, non-leaf nodes must create a channel for these recursive calls (we manually optimized the program so that all recursive calls from a node share a single reply channel, thus each internal node at the call tree creates only one boxed channel). They all together explain most of the differences between ABCL/f and C++. The performance of ABCL/f should become much closer to C++ when we implement a better code generator that applies unboxed channels to pointer data.

The baseline C++ program allocates the result of recursive calls on heap, only to return the result to the caller. This is clearly unnecessary. We wrote an optimized version that returns the result of recursive calls on stack. In this program, no heap allocations occur during the force calculation phase. "C++ (stack)" shows the result of this version, which was about 30 % faster than the baseline version.

Both sequential and parallel versions of CKY in ABCL/f run about 2.0 times slower than the baseline C++ program. Again, a large overhead comes from communication via boxed channels. Moreover, the problem is slightly harder than in BH. CKY builds a large matrix (CKY matrix) during parsing a sentence. Parse trees are constructed in a bottom up fashion; parse trees for shorter sub-sentences are built first and then combined to form parse trees for longer sub-sentences. The C++ program naturally achieves this bottom up behavior by building parse trees one after the other. The parallel ABCL/f program overlaps construction of parse trees for smaller sentences with that for longer sentences as much as possible. Hence threads which build a parse tree must synchronize with threads that produce its sub-trees. In the source code level, the C++ program obtains sub-trees simply by array references, whereas the ABCL/f program by method calls to concurrent objects that implement synchronization between the producer and consumers. Worse, since this method returns a pointer, the result of a method call must be communicated via a boxed channel. Again, we expect

a significant improvement is possible when we implement unboxed channels for pointer data.

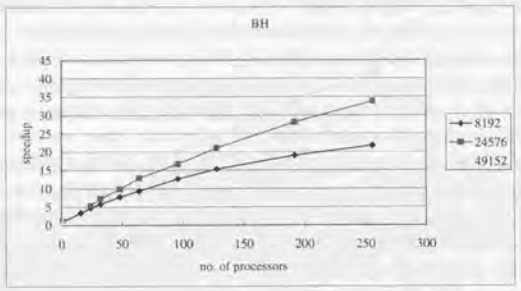
Performance of CKY is improved by customizing memory allocator for the particular lifetime distribution of parse trees. That is, all intermediate parse-trees for a sentence remain live until near the end of parsing the sentence, and they become dead, *en messe*, when parsing is finished. Hence, instead of requesting memory from a general memory allocator on a datum by datum basis, we can request a large block and allocate memory for parse trees from the block. They are simply recycled when we finish one sentence. Performance of C++ program with this optimized allocator is shown as C++ (fast alloc).

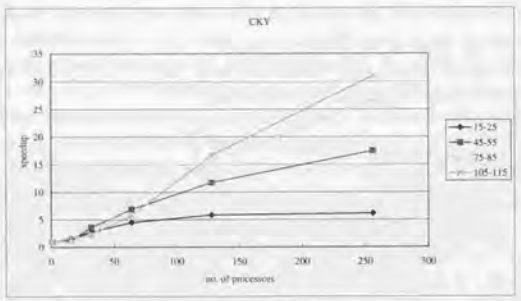
The above two applications suffer from the overhead of communication via boxed channel. RNA does not use channels of floating point or channels of pointer data in its kernel, and thus exhibits performance very close to C++. The overhead of the sequential version is in fact negligible. The parallel version incurs an overhead for performing asynchronous recursive calls until a certain depth of the call tree. Making future calls, *per se*, do not make any difference. The overhead is incurred because reply channels for the future calls are stored in cons cells and therefore boxed. In this benchmark, there were 65,000 future calls out of 1,300,000 total calls. That is, one out of twenty calls makes the reply channel boxed and performs an additional heap allocation for a cons cell.

6.2 Speed-up

Figure 6.2 shows speed-ups obtained on AP1000+ for various problem sizes. The baseline is the estimated time of the parallel ABCL/f binary executed on one processor of AP1000+. The single processor execution time on AP1000+ is estimated from the execution time on a faster workstation (UltraSparc 167 Mhz) and the ratio between those two processors in a small problem. For some applications, we were unable to directly measure the single processor execution time on AP1000+, because they take too long time or run out of memory (16 MB).

For BH, we set the number of particles to 8,192, 24,576, and 49,152 particles. On 256 processors, they correspond to having 32, 96, and 192 particles on each processor, respectively. When we have 49,152 particles,





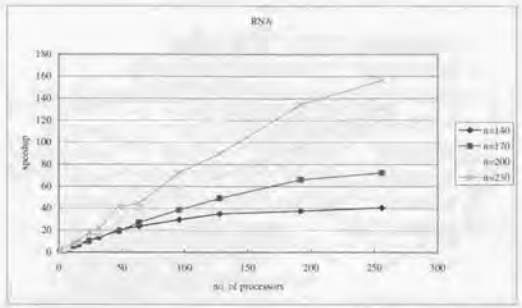


Figure 6.2: Speedup on AP1000+. RNA is quite scalable especially for large problem sizes. See the main text for performance limiting factors.

we observe 42 times speed-up on 256 processors, which is admittedly much lower than it ought to be. To understand the source of inefficiencies and how could it be improved, we analyze where time goes on various numbers of processors. Figure 6.3 breaks down the execution time of the force calculation phase into four categories, namely, busy, overhead, GC, and idle. The y-axis refers to the number of processors and *times are totaled over all the processors*. "Busy" refers to the time spent on user program, including calculation and replication, "overhead" the time spent on communication (send and receive) and context switches (block and resume), and "GC" the time spent on local/global GC (including idle time during global GC). "Idle" literally means the idle time, excluding idle time during global GC. First of all, the busy part clearly includes a portion that is proportional to the number of processors. The overhead part is also roughly proportional to the number of processors. This is because *each* processor must replicate a part of the BH tree, and thus the total amount of replications increases as we have more processors. The graph indicates that this poor scalability could be alleviated by optimizing communication layer or by having a (possibly built-in) more optimized layer for software caching. Next, there is a large

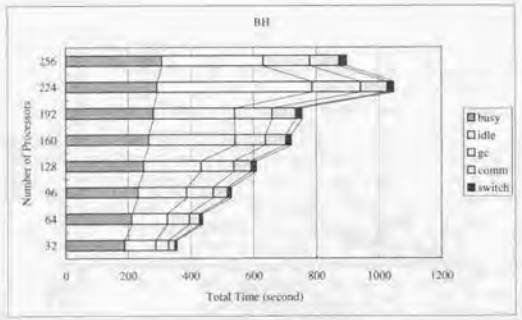
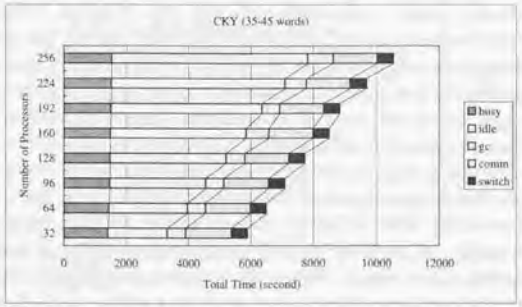


Figure 6.3: The breakdown of the execution time of BH into busy, overhead, GC, and idle. Times are totaled over all the processors. The amount of work (busy and overhead) noticeably increases as the number of processors. This is because each processor must replicate a part of BH-nodes. Idle time is also large due to inadequate load balancing method we currently implement.

fraction of idle time. This comes from an inadequate load balancing method we currently implement. We sort particles according to the Morton ordering [75] and assign *the same number of particles* to each processor. Since the density of particles significantly varies from one place to another, assigning the same number of particles cause processors assigned to a dense region to be heavily loaded. We examined an application profile and observed that most idle times appear at the end of the force calculation, confirming that the idle time is due to load imbalance, rather than latency of synchronous communication.

For CKY, we tested sentences of various lengths, namely, 35-45, 65-75, and 95-105. Longer the sentences are, better speed-up we achieved. Breakdown of the application time is also shown for the short sentences (35-45) and the long sentences (95-105) in Figure 6.4. Unlike BH, the amount of work (busy, GC, and overhead) is essentially constant.

In general, CKY exhibits an even severer speed-up than BH, and it is much harder to establish a simple performance model for it. Our analysis



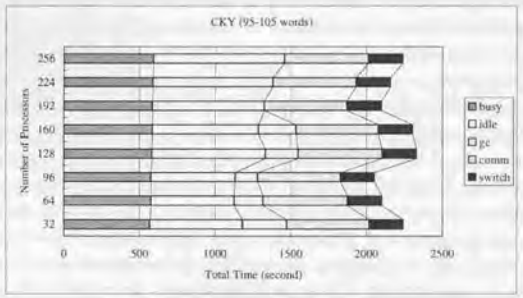


Figure 6.4: The breakdown of the execution time of CKY into busy, overhead, GC, and idle. Times are totaled over all the processors. Unlike BH, the amount of work is constant. The limiting factors are overhead for communication and idle time due to a severe critical path length (for short sentences) and load imbalances.

so far indicates that speed-up is limited by several factors. First, a critical path inherent in the algorithm limits speed-up for short sentences. Recall that parsing proceeds from bottom to top. To completely finish parsing the entire sentence (call it $w_1 w_2 \cdots w_n$), we must wait for the completion of parsing both sub-sentences $w_1 w_2 \cdots w_{n-1}$ and $w_2 w_3 \cdots w_n$ because there are possibilities that parse trees for those sub-sentences constitute an entire parse tree. In general, to complete parsing a sentence of length n, we must wait for the completion of parsing its sub-sentences of length $n-1$, and then examine if they constitute a parse tree of the entire sentence. This places a severe upper bound on the achievable performance, particularly for short sentences. Refer to [54] for a more detailed analysis. From an application profile, we attribute the poor speed-up for sentences of length 35-45 to the critical path length. Second, for sentences of any length, communication overhead between a thread that produces a sub-tree and other threads that read it places an upper bound on the efficiency (the ratio between the achieved speed-up and the ideal speed-up). Our CKY implementation creates a thread for any sub-sentence of the given sentence. That is, for any p, q such that $1 \leq p < q \leq n$, we create a thread which builds parse trees for sub-sentence $w_p \cdots w_q$. The thread that is assigned to this sentence must receive results from threads that produce parse trees for its sub-sentences. The ratio between the cost of this communication + associated context switches and the cost of useful computation (*i.e.,* combining parse trees to form a larger tree) limits the efficiency. Finally, load imbalance limits processor utilization. From the lower graph in Figure 6.4, we observe that there is a significant amount of idle time even for long sentences where the critical path should not be a problem. By profiling the amount of work done by each processor, we attribute this to load imbalance. Since the work performed by each thread is highly uneven and depends on input, and the number of threads at each processor is at most 40 or so on 256 processors, the total work performed by each processor is unlikely to be balanced enough.

RNA just exhibits an encouraging speed-up, at least for large problems. This is because RNA is a simple parallel tree search problem where threads do not synchronize and communication is not frequent. n refers to the size of the problem and the size of the search space is exponential to n. A similar breakdown of the application time for the maximum problem size ($n = 230$) is shown in Figure 6.5. The amount of work is approximately constant, but

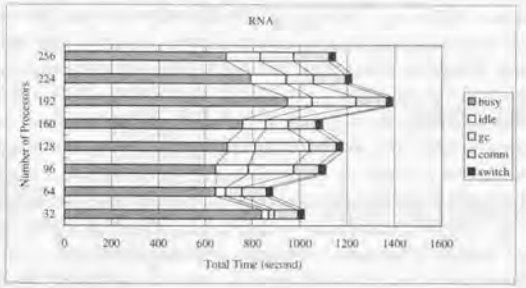


Figure 6.5: The breakdown of the execution time of RNA into busy, overhead, GC, and idle. The amount of work slightly varies unpredictably, probably due to the inherent indeterminacy of the application.

slightly varies unpredictably. This is probably due to the inherent indeterminacy of the application. How much pruning occurs depends on timing and may differ from one invocation to another.

Chapter 7

Conclusion and Future Work

The main contribution of this thesis is efficient and reusable implementation of multithreading and garbage collection and empirical results obtained by building a new programming language ABCL/f and writing applications in it.

Multithreading mechanisms have been studied in many contexts and by various approaches, including hardware solutions [53, 61], compiler-centric approaches [6, 64, 65, 73], and runtime-centric approaches (which are, of course, not exclusive with each other). Our study clearly falls into the runtime-centric approach. Previous work, Lazy Task Creation (LTC) in particular, has proposed the basic execution mechanisms in this area and it has been studied in the context of a parallel Lisp on shared-memory machines. Additional contributions of this work are two folds. First, LTC has been studied with relatively small and mostly functional applications, assuming hardware-supported shared heaps,[1] while we study our mechanism in the context of larger and more complex applications on distributed-memory machines. As an environment in which multithreading is studied, mostly functional programs with hardware supported shared heap was somewhat less severe than our setting. When data are mostly read and remote data can be fetched quickly by hardware, one does not have to seriously worry about switch cost or switch frequency. Having object-oriented applications in which data are frequently locked with fine granularity and distributed

[1] Feeley's message passing protocol [29] does not assume shared stack, but assumes shared heap.

memory machines in which stalling the entire processor on a remote access is undesirable and sometimes difficult to implement, empirical studies were needed to verify that fine-grain multithreading is really feasible. Second, previous runtime-centric approaches are, although in principle applicable to other languages, not readily sharable by other language implementers. Previous runtime-centric approaches were in fact compiler-centric, in the sense that extensive cooperation was required from the code generator. As a consequence, it has not been clear to which extent LTC-like mechanisms can be efficiently implemented in such a way that strictly preserves the sequential calling standard as well as many compilers and support tools built on top of it. Our multithreading mechanism maximally exploits information already present in standard C stack frames as much as possible and has been successfully implemented on two different platforms (Sparc and Alpha). Thanks to such runtime mechanism, we were able to implement a programming language whose sequential speed is as fast as C and whose thread creation overhead is as low as LTC, without re-designing runtime data format, calling convention, and code generator which conforms to the convention from scratch. Performance studies so far indicate that supporting multithreading languages in this way is indeed feasible. Assuming data distribution with reasonable amount of locality or appropriate replication strategy, multithreading overhead (overhead for preparing potential blocking and overhead for thread switch) is never significant, as shown in Chapter 2.

Garbage collections have also been studied in broad contexts. Even when we restrict our attention to ones on distributed memory environments, variety of algorithms have been proposed in the literature. Our garbage collection algorithm is very straightforward, compared to many previous algorithms that deal with issues such as faulty processes or lost messages. Most of previous studies, however, only present algorithms and have not been implemented on real machines. Performance is often studied only qualitatively. Our primary contribution to the community is an empirical and qualitative performance study with a reasonable heap expansion policy clearly mentioned. Garbage collection overhead is necessarily relative to behavior of the application and how much memory is allowed to use, thus performance studies without reasonable heap expansion policies may not be reproducible when the amount of available memory differs. We show that, under a modest heap expansion policy that preserves that of the original Boehm & Weiser's

collector, the garbage collection overhead is in the ballpark of that in the sequential program. We also show that local collections should be typically invoked synchronously on all the processors, at least in our experimental settings (256 processors and heap expansion policies stated in Chapter 3), despite its synchronization cost and potential extra work. In essence, this is a restatement of the "co-scheduling benefit" of local collectors, but has been overlooked by the community, because we have an intuition that a global synchronization is expensive and does not scale. Furthermore, an adaptive scheduling strategy that selects the appropriate strategy has been developed. Our experiments so far indicate that it selects the right strategy when one is clearly better than the other.

Applications have been written in ABCL/f and their performances have been compared to equivalent sequential C++ programs. The overhead of ABCL/f on a single processor workstation varies from 30 % to 110 %. A large source of the sequential overhead was a current limitation of the compiler that always boxes reply channels of pointer data. This will certainly be fixed in the future. Another, more serious problem is allocation and (local) collection overhead. In spite of the allocation performance of Boehm & Weiser's collector that is superior to usual malloc + free, C++ programs can manually customize allocation performance by taking advantage of the application-specific allocation behavior and lifetime distribution. There may even be cases where data can be allocated on stack. We do not have an immediate answer to this problem. Applications exhibit from 40 to 160 times speed-up on 256 processors and speed-up is often limited by communication overhead. This is partially due to current implementation that favors portability across different message passing interfaces. Reengineering communication code will produce a better result.

Many issues should be investigated more extensively. Below we list only ones that will immediately follow the present work.

Portability Guarantee of StackThreads: Although StackThreads has been shown to add little restrictions to the current C compilers and it has been ported to Sparc and Alpha, we wish to guarantee the portability of the approach in any reasonable calling conventions for C or C++, hopefully under more relaxed assumptions. We roughly made three assumptions, namely, ability to thread epilogue code sequences through a call chain, ability to traverse the chain of stack

frames, and mobility of stack frames. First two assumptions would be most conveniently satisfied by a set of simple extensions to C that provide information about the current procedure such as the size of parameters and the offset where return address is saved. Such extensions may not be available on existing compilers, however. In a short term, a more practical approach will be providing procedure descriptor by post-processing assembly code generated by C compilers. Such descriptors will be a modest extension to exception handling mechanisms provided on some operating systems such as Digital UNIX and IRIX and may hopefully be incorporated into a standard. Mobility of stack frames seems to be a valid assumption as long as aggregate data are not allocated on stack. For StackThreads to be more useful, however, we wish to guarantee the safety of code that allocates aggregate data on stack, as long as the address is not explicitly taken. Such guarantee seems to necessarily require compiler extensions. Right now, we do not have a better alternative to this problem.

Study of the adaptive local collection strategy in various settings: While the adaptive collection strategy investigated in Section 3.4.4 chooses the right strategy where one strategy is clearly better than the other, it was not substantially better than a simpler "always-synchronous" approach. In fact, independent local collections have (if any) little gain over synchronous ones, thus the adaptive strategy can always resort to the synchronous strategy whenever the right strategy is not clear. This may not be the case in other settings. First, we suspect that this was the case partially because of our heap expansion policy. Our heap expansion policy assumes that if any processor has expanded its local heap up to M, it is reasonable for any processor that has much smaller heap size to expand its local heap up to around M. This policy avoids too frequent local collections when many processors are simultaneously expanding heap sizes. On the other hand, this may still expand heap too aggressively, particularly when live data among processors are highly unbalanced. Right now, we do not have an alternative expansion policy that is less aggressive under unbalanced live data distribution and does not lead to unreasonably frequent collections when processors are simultaneously expanding their heaps.

Assuming the presence of such a policy, the policy would be likely to favor independent collections more than the current policy does, and thus the importance of the adaptive strategy would accordingly increase. Second, the advantage of synchronous collections may be reduced in multiprogrammed environments (network of workstations) where synchronization delays may be more unpredictable. Right now, we simply do not know how do they perform on today's commodity operating systems in which processes are scheduled independently. While future operating systems for high performance workstation clusters will support some form of co-scheduling for synchronous SPMD applications, the advantage may be smaller than in dedicated parallel computers such as AP1000+.

Comparison to reference counting methods: Our experimental results, which favor synchronized collections over independent ones, partially contradicted previous scalability criteria of collectors on large-scale systems. We suspect naive reference counting schemes suffer from the same problem with the independent collection scheme and tend to require a larger amount of memory than stop-the-world type collector does. It will be fruitful to examine this conjecture through experiments and explore the possibility for a combined strategy. The combined strategy reclaims small and locally shared data incrementally by reference counting and reclaims large and globally shared linked data structure by global mark & sweep, with one stroke.

Dynamic load balancing: One thing that is overlooked in this work despite of its importance is dynamic load balancing, where a computation migrates to another processor in the middle of it. In addition to performance considerations, implementing migration was already hard a problem in the context of our work. It would require precise identification of pointers in a C stack frame or a lower level software support for shared address space. As large-scale shared-memory machines are getting popular and more widely available, assuming hardware support for shared-memory will not become a severe restriction in the future. Hence, we wish to explore the possibility for implementing dynamic load balancing again with existing sequential C/C++ compilers. The execution mechanism will naturally be similar to Lazy Task Creation,

but again the problem is how to make it implementable under the current C stack frames and calling conventions. In the presence of callee-save registers, it seems unavoidable that a task stealing requires some cooperation from the victim. Hence, the mechanism will be based on the message passing protocol investigated by Feeley [29]. Here we vaguely describe the task stealing mechanism under standard C stack frames and calling conventions. When a victim picks up a task-stealing request, it unwinds stack frames using the epilogue code threading until an appropriate fork point. At that point, we transfer control to a special routine that handles the request on a separate stack. The handler copies the stolen continuation and makes it available to the requesting processor. It then modifies the original continuation so that when control reaches to the continuation, the stolen frames are simply discarded, again by epilogue code threading. The handler finally resumes the original computation.

Bibliography

[1] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Collection schemes for distributed garbage. In *Proceedings of International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 43–81. Springer-Verlag, 1992.

[2] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software, Practice & Experience*, 19(2):171–183, February 1989.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Department of Computer Science, Princeton University, 1994.

[6] Takuya Araki and Hidehiko Tanaka. A static granularity optimization method of a comitted-choice language fleng. *Transactions of Information Processing Society of Japan*, 1997. (to appear).

[7] Yariv Aridor. *An Efficient Software Environment for Implicit Parallel Programming with a Multi-Paradigm Language.* PhD thesis, the Senate of Tel-Aviv University, 1995.

[8] W. C. Athas and C. L. Seitz. Cantor user report version 2.0. Technical report, Computer Science Department, California Institute of Technology, 1987.

[9] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[10] David I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, number 258, 259 in Lecture Notes in Computer Science, pages 176–187, Springer-Verlag, 1987.

[11] Andrew Birrel, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical Report 116, Digital Systems Research Center, 1993.

[12] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of Fourteenth Symposium on Operating Systems Principles (SOSP)*, pages 217–230, 1993.

[13] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing*, pages 588–597, 1993.

[14] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel langauge. *Scientific Computing*, 2(3), 1993.

[15] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[16] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[17] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. The MIT Press, 1993.

[18] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[19] Andrew A. Chien. *Concurrent Aggregates (CA)*. MIT Press, 1991.

[20] Andrew. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. In *Proceedings of the Second International Symposium on Object Technologies for Advanced Software*, 1996.

[21] Takashi Chikayama. Personal Communication.

[22] Eric C. Cooper. Adding threads to standard ML. Technical Report 90-186, Carnegie Mellon University, Pittsburgh, December 1990.

[23] Eric C. Cooper and Richard P. Draves. *C Threads*. Department of Computer Science, Carnegie Mellon University, 1987.

[24] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 166–175, 1991.

[25] Digital Equipment Corporation. *DEC OSF/1 Assembly Language Programmer's Guide*, 1994.

[26] Digital Equipment Corporation. *DEC OSF/1 Calling Standard for AXP Systems*, 1994.

[27] J. P. Dumas and J. Ninio. Efficient algorithms for folding and computing nucleic acid sequences. *Nucleic Acids Res.*, 10(1):197–206, 1982.

[28] Marc Feeley. Lazy remote procedure call and its implementation in a parallel variant of C. In *Proceedings of International Workshop on Parallel Symbolic Languages and Systems*, number 1068 in Lecture Notes in Computer Science, pages 3–21. Springer-Verlag, 1995.

[29] Mark Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[30] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed strage reclamation scheme. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 313–321, 1989.

[31] Seth Copen Goldstein, Klaus Erik Schauser, and David Culler. Enabling primitives for compiling parallel languages. In *Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995.

[32] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulation of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing '94*, pages 439–448, 1994.

[33] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, April 1985.

[34] Kenichi Hayashi, Tunehisa Doi, Takeshi Horie, Yoichi Koyanagi, Osamu Shiraki, Nobutaka Imamura, Toshiyuki Shimizu, Hiroaki Ishihata, and Tatsuya Shindo. AP1000+: Architectural support of PUT/GET interface for parallelizing compiler. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, pages 196–207, 1994.

[35] Kazuo Hiyane. Generation of a set of pareto-optimal solutions for multiobjective optimization by parallel genetic algorithms and its quantitative estimation. In *9th SICE Symposium on Decentralized Autonomous Systems*, pages 295–300, 1997. (In Japanese).

[36] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[37] John Hughes. A distributed garbage collection algorithm. In *Proceedings of the ACM Conference on Functional Programming Languages*

and Computer Architecture, number 201 in Lecture Notes in Computer Science, pages 256–272. Springer-Verlag, 1985.

[38] Nobuyuki Ichiyoshi, Kazuaki Rokusawa, Katsuto Nakajima, and Yu Inamura. A new external reference management and distributed unification for KL1. In *New Generation Computing*, volume 7, pages 159–177. Springer-Verlag, 1990.

[39] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[40] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 103–115. Springer-Verlag, 1992.

[41] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In *Proceedings of Supercomputing '94*, pages 79–88, 1994.

[42] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[43] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, 1965.

[44] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. SunSoft Press, 1996.

[45] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.

[46] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–58, 1992.

[47] Eric Mohr. Distillations of dynamic partitioning experience. In Robert H. Halstead, Jr. and Takayasu Ito, editors, *Proceedings of Parallel Symbolic Computing: Langugages, Systems, and Applications*, volume 748 of *Lecture Notes in Computer Science*, pages 88–93. Springer-Verlag, 1992.

[48] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A techinque for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[49] Akihiro Nakaya, Kenji Yamamoto, and Akinori Yonezawa. RNA secondary structure prediction using highly parallel computers. *Comput. Applic. Biosci. (CABIOS)*, 11, 1995.

[50] Anton Nijholt. Parallel approaches to context-free language parsing. In *Parallel Natural Language Processing*, pages 135–167. Ablex Publishing Corporation, 1994.

[51] Rishiyur S. Nikhil. Parallel symbolic computing in Cid. In Takayasu Ito, Robert H. Halstead, Jr., and Christian Queinnec, editors, *Proceedings of International Workshop on Parallel Symbolic Languages and Systems*, number 1068 in Lecture Notes in Computer Science, pages 217–242. Springer-Verlag, 1995.

[52] Rishiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. Technical report, Massachusetts Instituted of Technology, Cambridge, 1990.

[53] Rishiyur S. Nikhil, Greg. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *The 19th Annual International Symposium on Computer Architecture*, pages 156–167, 1992.

[54] Takashi Ninomiya, Kenjiro Taura, Kentaro Torisawa, and Jun'ichi Tsujii. A scalable implementation of parallel CKY algorithm in concurrent object-oriented language ABCL/f. In *Proceedings of JSSST Workshop on Object-Oriented Computing (WOOC)*, 1997. (in Japanese).

[55] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *Proceedings of Parallel Architectures and Lan-*

guages Europe, number 505, 506 in Lecture Notes in Computer Science, pages 150–165. Springer-Verlag, 1991.

[56] Plainfossé and Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[57] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Supercomputing '95*, 1995.

[58] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[59] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, 1995.

[60] Kazuaki Rokusawa and Nobuyuki Ichiyoshi. Evaluation of remote reference management in a distributed KL1 implementation. In *IPSJ SIG Notes 96-PRO-8 (Proceedings of Summer Workshop on Parallel Processing)*, pages 13–18, 1996. (in Japanese).

[61] Shuichi Sakai, Yoshinori Yamaguchi, and Kei Hiraki. An architecture of a dataflow single chip processor. In *The 16th Annual International Symposium on Computer Architecture*, pages 46–53, 1989.

[62] John Sargeant. United functions and objects: An overview. Technical report, Department of Computer Science, University of Manchester, 1993.

[63] John Sargeant. Uniting functional and object-oriented programming. In Shojiro Nishio and Akinori Yonezawa, editors, *Proceedings of First JSSST International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1993.

[64] Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation constraint partitioning — a new algorithm for partitioning non-strict programs into sequential threads. In *Conference Record on POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 259–272, 1995.

[65] Klaus Erik Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution*. PhD thesis, Department of Electrical Engineering and Computer Science, 1996.

[66] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, 1994.

[67] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–87. The MIT Press, 1991.

[68] SPARC International Inc. *The SPARC Architecture Manual*, 1992.

[69] Guy L. Steel, Jr. *Common Lisp, The Language, Second Edition*. Digital Press, 1992.

[70] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 218–228, 1993.

[71] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Proceedings of Workshop on Theory and Practice of Parallel Programming*, number 907 in Lecture Notes on Computer Science, pages 121–136. Springer Verlag, 1994.

[72] Kenjiro Taura and Akinori Yonezawa. Schematic: A concurrent object-oriented extension to scheme. In *Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, number 1107 in Lecture Notes in Computer Science, pages 59–82. Springer-Verlag, 1996.

[73] Kenneth R. Traub, David E. Culler, and Klaus E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 324–334, San Francisco, California, June 1992.

[74] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 208–217, 1993.

[75] Michael S. Warren and John K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing '92*, pages 570–576, 1992.

[76] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing '93*, pages 12–21, 1993.

[77] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, number 258, 259 in Lecture Notes in Computer Science, pages 432–443, Springer-Verlag, 1987.

[78] William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. PRELUDE: A system for portable parallel software. Technical Report MIT/LCS/TR-519, Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.

[79] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[80] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, pages 258–268, 1986.

Appendix A

Description of Benchmark Applications

For BH, CKY, and RNA, we describe the problem, basic algorithm, parallelization and its description in ABCL/f, and behavior of the parallel program.

A.1 BH

A.1.1 Problem

Given initial velocities and positions of particles, simulate motions of them. Any pair of two particles interacts with each other via Newtonian force:

$$F = G\frac{mM}{r^2}$$

where r is the distance between the two particles and m and M are masses of them.

A.1.2 Basic Algorithm

Since naive algorithm that calculates interaction between all pairs of particles takes $O(n^2)$, we use an approximation method widely known as Barnes-Hut method [9], or BH-method in short. When we calculate a force exerted on a particle, we regard a set of particles whose center of gravity is far

from the particle as a single (virtual) particle located on the center of gravity. More precisely, let r be the distance between the particle in question and the center of gravity of the particles, and let d be the diameter of the particles. We regard the set of particles as a single particle if:

$$\frac{d}{r} < \theta,$$

where (θ is a constant (1.0 in our experiment).

To enable this approximation, we construct an Oct-tree each node of which represents a cubical cell in the simulation space. The root of the tree represents a cell that is big enough to contain the entire particles. The (direct) children of a node represent subdivisions of the parent, derived by cutting the parent cell into eight equally sized sub-cells. A node has children when two or more particles fall into the cell represented by the node. In other words, at leaves of a BH-tree are nodes that contain at most one particle. The tree is constructed at each step from scratch. To summarize, each step of a simulation proceeds as follows:

Tree-construction phase: constructs a BH-tree, so that leaves contain at most one particle,

Augment phase: augments every node of the tree with the center of gravity and total mass of the particles contained in it,

Force-calculation phase: calculates force exerted on each particle, and

Update phase: updates position and velocity of each particle.

The sequential algorithm for the tree construction phase begins with an empty tree that holds no particles. From this initial state, we 'load' particles one by one from the root of the tree. Whenever the second particle is loaded into a node, which currently contains only one particle, we create children of the node and load the particle to the appropriate child. Subsequent particles loaded into a cell that already has children are just forwarded to the appropriate child. Once the tree has been constructed, the augment phase sets the center of gravity and the total mass of the particles contained in each node by a depth-first traversal of the tree. The main procedure of the force-calculation phase is a recursive procedure that takes a position of a particle and a node of a BH-tree as parameters. It tries to calculate an

acceleration that the particles contained in the node exert on the particle. It simply returns the Newtonian acceleration if the node is either a leaf or an internal node that meets the approximation criterion described above. Otherwise it recursively applies the procedure to all its children and sums up the results. When all the accelerations are calculated, the update phase updates velocities and positions of all particles.

A.1.3 Description in ABCL/f and Parallelization

We represent a node of a BH-tree by a concurrent object. The tree-construction phase is implemented by an update method (*i.e.*, a method defined by `defmethod!`) for node objects, which takes a particle as the parameter and loads the particle into the tree. The method updates self when it was originally a leaf, or otherwise simply forwards the given particle to the appropriate child. The augment phase is implemented by a recursive update method that traverses the tree and augments every node by its total mass and the center of gravity. The force-calculation is implemented by a read-only method that recursively traverses the tree until leaves or nodes that meet the approximation criterion.

The source of parallelism can be easily identified. Force-calculation phase can calculate forces of all particles in parallel. Augment phase can traverse different part of the tree in parallel. Less obviously, the tree construction phase can load all particles in parallel, as long as method invocations on a leaf node are properly serialized. A difficult part is how to remove bottleneck and obtain a load balance.

The most time-consuming phase is force-calculation, which occupies more than 80% of the total execution time. Tree-construction approximately takes 15% and other two phases the rest 5% . While force-calculation is dominant, it is yet important to achieve reasonable speed-up for other phases; for example, if only the force-calculation would be parallelized, the total execution time would be at best reduced to 20% of the sequential execution time.

In the original description, neither the force-calculation nor the tree-construction achieves satisfactory speed-up, because almost all recursive calls become remote, introducing large overhead. Moreover, nodes near the root are much more frequently accessed than nodes near leaves are, in-

troducing a significant load imbalance. Caching node objects appropriately alleviate these problems. In our experiments, we manually replicate objects in ABCL/f level. Each processor maintains a hash table that associates remote objects to their local copies.

The caching protocols take advantage of application-specific knowledge about access patterns. In the force-calculation phase, nodes are read-only. Hence we simply replicate objects without any provision for future invalidation. A processor calculates forces to its particles, one particle after the other. A calculation for a particle utilizes replicas created by previous particles. The entire cache is discarded at the end of the phase.

The protocol for the tree construction phase is less straightforward. We take advantage of the fact that a node becomes read-only after it once becomes a non-leaf node. The protocol works as follows. Before a processor calls a method on a BH-node, the processor first looks up its cache. If the replica is in the cache, it simply calls the method with the replica as the receiver object. Otherwise, it asks the BH-node to create a copy of itself on the requesting processor and returns it to the requesting processor, *as long as the BH node has already become a non leaf node*. What happens if the node is still a leaf? It simply returns self to the requesting processor. Either case, the requesting processor invokes a method on the returned object, which may be a local copy or the original object. Again, the entire cache is discarded at the end of the phase, hence the protocol does not have any provision for invalidation.

One remaining problem is how to assign particles to processors. An obvious requirement is load balance. Another, less obvious one is locality, by which we mean particles physically close to each other must be co-located on the same processor as much as possible. This comes from the access patterns implied by the above approximation criterion. Since particles close to each other access a similar set of BH-nodes, co-locating such particles better exploit the above introduced software caches. [76] describes a method that achieves both load balance and locality. We only partially implement this method, achieving only locality. As for the load balance, we currently assign the same number of particles to each processor.

A.1.4 Behavior and Performance Limiting Factors

We only describe the behavior of the force-calculation phase, which dominates the overall behavior. Since each processor sequentially calculates a force exerted on a particle, there is no intra-processor parallelism. A processor stalls when a copy of the receiver object is not in the cache. In an early stage of a force-calculation phase, processors stall very frequently. Each access to a BH-node that has never been accessed by the processor in this step causes the processor to stall. Stalls become less frequent as caches are getting filled. This synchronous behavior makes this application very sensitive to latency.

Performance of this algorithm is currently limited by the following factors:

Replication Overhead: Since each processor starts with an empty cache and fills it with necessary data on demand, the total work carried out by a processor is force-calculation + copying part of a BH-tree necessary for the force-calculation. The replication overhead actually involves communication overhead, switch overhead, and object creation overhead. It accounts a large fraction of the total execution time, as we have seen in Section 6.2.

Load Imbalance: Since we assign the same number of particles to each processor, processors being in charge of dense regions tend to be heavily loaded. This results in load imbalance we have observed in Section 6.2

A.2 CKY

A.2.1 Problem

Given a context free grammar (CFG) in its Chomsky Normal Form and an input sentence, judge if the sentence is produced by the CFG, and if it is, leave sufficient information to reproduce the derivation tree (parse tree).

In Chomsky Normal Form, a rule is either a *lexical* rule:

$$a \to w,$$

where a is a non-terminal and w a terminal, or a *production* rule:

$$a \rightarrow b\,c,$$

where a, b, and c are non-terminal symbols. That is, the right hand side of a production rule is binary.

The actual benchmark first copies the grammar to all the processors and feeds many input sequences one after another. We parallelize parsing of a single sentence and do not overlap processes for multiple sentences. We also exclude the time to broadcast the grammar from the benchmark.

A.2.2 Basic Algorithm

We use CKY algorithm [43]. Let $w_1 w_2 \cdots w_n$ be the input sentence, $S_{i,j}$ a set of non-terminal symbols that derive sub-sentence $w_{i+1} \cdots w_j$. The problem now is to find $S_{0,n}$, a set of non-terminal symbols that derive the whole sentence, and check if it contains the start symbol. The CKY algorithm calculates $\{S_{i,j}\}$ $(0 \leq i < j \leq n)$ in a bottom up manner, *i.e.*, from $S_{i,j}$s for shorter sentences to ones for longer sentences; it first calculates $S_{i-1,i}$ for all $1 \leq i \leq n$, using lexical rules. By the definition of $S_{i,j}$, $S_{i-1,i}$ refers to a set of non-terminal symbols that derive sub-sentence w_i. Hence, for each a such that $a \rightarrow w_i$, we include a in $S_{i-1,i}$. Once all $S_{i-1,i}$ $(1 \leq i \leq n)$ have been calculated, we can calculate $S_{i-2,i}$ $(2 \leq i \leq n)$. To calculate an $S_{i-2,i}$, we find all production rules $a \rightarrow b\,c$, such that $b \in S_{i-2,i-1}$ and $c \in S_{i-1,i}$ and include a in $S_{i-2,i}$. That is, if b derives sub-sentence w_{i-1}, c derives w_i, and there is a production rule $a \rightarrow b\,c$, then we have that a derives $w_{i-1} w_i$.

In general, to obtain an $S_{i,j}$ where $j > i+1$, we find all combinations of an index k $(i < k < j)$ and a production rule $a \rightarrow b\,c$, such that $b \in S_{i,k}$, and $c \in S_{k,j}$, and include all such a in $S_{i,j}$. That is, if b derives sub-sentence $w_{i+1} \cdots w_k$, c derives $w_{k+1} \cdots w_j$, and there is a production rule $a \rightarrow b\,c$, then we have that a derives $w_{i+1} \cdots w_j$. To describe how to compute $S_{i,j}$ more procedurally, for each k $(i < k < j)$, we generate all pairs $\langle x, y \rangle$ such that $x \in S_{i,k}$, and $y \in S_{k,j}$, and for each pair, we consult the grammar to find rules whose right hand side is x and y. Include the left hand side of all such rules in $S_{i,j}$, removing duplications (Figure A.1).

Along with $S_{i,j}$, we build data structure $E_{i,j}$, which record derivation trees. $E_{i,j}$ are not consulted during parsing, but necessary to later reproduce


```
 1: /* calculate a set of non-terminal symbols, each symbol in which
 2:    derives sub-sentence w_{i+1} ... w_j. */
 3: calc_Sij (i, j)
 4: {
 5:   r = ∅;
 6:   foreach k (i < k < j)
 7:     foreach x ∈ S_{i,k}
 8:       foreach y ∈ S_{k,j}
 9:         r = r ∪ { a | a → x y ∈ P };
10:   return r;
11: }
12:
13: /* return SUCCESS if non-terminal symbol S derives
14:    w_1 ... w_n. return FAIL otherwise.*/
15: CKY (w_1 ... w_n, S)
16: {
17:   foreach i (1 < i < n)
18:     S_{i-1,i} = { a | a → w_i ∈ L };
19:   for (l = 2; l <= n; l++) {
20:     for (i = 0, j = l; j <= n; i++, j++) {
21:       S_{i,j} = calc_Sij (i,j);
22:     }
23:   }
24:   if (S ∈ S_{0,n}) return SUCCESS;
25:   else return FAIL;
26: }
```

Figure A.1: CKY algorithm. `CKY` takes a sentence $w_1 \cdots w_n$ and a start symbol S as parameters and returns whether or not S derives $w_1 \cdots w_n$ in a given grammar. In the program, P refers to the set of production rules and L the set of lexical rules. The algorithm first calculates $S_{i,j}$ for sub-sentences of length 1 (*i.e.*, $S_{i-1,i}$) and proceeds to longer sentences. An iteration of the `for` loop at line 19 calculates $S_{i,j}$ for sub-sentences of length l.

the derivation process as necessary. For each symbol $a \in S_{i,j}$, we record k and the production rule used to include a in $S_{i,j}$.

As revealed from the above description, the process is bottom up in nature. More specifically, computing $S_{i,j}$ needs $S_{i,k}$ and $S_{k,j}$ for all k ($i < k < j$), thus $S_{i,k}$ and $S_{k,j}$ must be computed before $S_{i,j}$. A sequential algorithm satisfies this constraint by the order in which $S_{i,j}$ are computed. It calculates $S_{i,j}$ sequentially, from ones that have smaller $j - i$ to ones that have larger $j - i$.

A.2.3 Description in ABCL/f and Parallelization

We fork a thread for each $S_{i,j}$ using the future construct of ABCL/f. The thread that calculates $S_{i,j}$ processes $j - i - 1$ pairs $\langle S_{i,k}, S_{k,j} \rangle$ ($i < k < j$) sequentially, waiting for $S_{i,k}$ and $S_{k,j}$ to be produced for each k. When a thread processes all pairs, it produces $S_{i,j}$, resuming threads waiting to consume it. To accomplish this producer-consumer synchronization, we define a cell object which has two methods, `put!` and `get!`. They are implemented using explicit reply channel in the manner described in Section 4.5.2.

Let us examine how much parallelism are there in this algorithm. Obviously, computations for all $S_{i,j}$s of the same length do not depend on each other, so they run fully in parallel. In other words, the inner loop at line 20 of Figure A.1 is a 'doall' loop. How about the outer loop? Since computation of an $S_{i,j}$ depends on computations of its sub-sentences of the form $S_{i,k}$ or $S_{k,j}$, it is not clear how much parallelim are there between iterations of the outer loop.

Fortunately, we can extract a significant amount of parallelism from the outer loop by carefully ordering the computation. When a thread calculate $S_{i,j}$ from $\{S_{i,k}\}$ and $\{S_{k,j}\}$ ($i < k < j$), we first process pairs that are likely to be produced early. To achieve this, we begin with k which is most close to $(i + j)/2$ and step towards both edges (i and j). For example, a thread that computes $S_{0,20}$ first processes pair $\langle S_{0,10}, S_{10,20} \rangle$, next $\langle S_{0,9}, S_{9,20} \rangle$ and $\langle S_{11,20}, S_{11,20} \rangle$, and so on. Both $S_{0,10}$ and $S_{10,20}$ will probably be produced much earlier than $S_{0,19}$ or $S_{1,20}$. Thus, there is significant overlap between computations for $S_{i,j}$s of different lengths. Roughly, threads that compute $S_{i,j}$ where $j - i = c$ overlap with threads that compute $S_{i,j}$ where $j - i > c/2$.

The amount of work carried out by each thread is neither constant nor

very predictable. In general, $S_{i,j}$ with larger $j - i$ represent more tasks than those with small $j - i$, because $S_{i,j}$ is computed by processing $j - i - 1$ pairs. Further details are, however, dependent on the input sentence. Threads are mapped onto processors so that each processor is in charge of approximately the same number of threads and threads with large $j - i$ are not assigned to the same processor. More specifically, we first allocate $S_{0,n}$ to processor 1, next $S_{0,n-1}$ and $S_{1,n}$ to processor 2 and 3, respectively, then $S_{0,n-2}, S_{1,n-1}$, and $S_{2,n}$ to processor 4, 5, and 6, respectively, and so on. The cell object that stores $S_{i,j}$ is co-located with the thread that produces it.

See [54] for more detailed description.

A.2.4 Behavior and Performance Limiting Factors

When a thread that computes $S_{i,j}$ processes a pair $\langle S_{i,k}, S_{k,j} \rangle$, it is likely to stall due to remote communication. Recall that $S_{i,k}$ and $S_{k,j}$ are co-located with the threads that compute them, and threads are mapped onto processors in a round-robin fashion. Thus, threads for $S_{i,j}$ and $S_{i,k}$ (or $S_{k,j}$) are unlikely to be mapped on the same processor. The latency involves not only the latency of remote communication, but also that of producer-consumer synchronization whose delay is unpredictable. The latency is partially (or hopefully completely) masked by other threads on the same processor. When we parse short sentences on many processors, it is unlikely to be; sentences of 30 words yield only $(30 \times 31)/2 = 465$ threads, which are not much larger than the maximum number of processors we have. On the other hand, if the length of the sentence is 100, we create $(100 \times 101)/2 = 5050$ threads, meaning that we have 20 threads on one processor even on 256 processors. In this case, we have a reasonable chance to utilize processors while one thread is waiting for a value from another thread. This application allocates a large amount of memory for $S_{i,j}$ and $E_{i,j}$. These data remain live until parsing the current sentence is finished.

There are several factors that currently limit the performance of this algorithm.

Overhead: When a thread processes a pair $\langle S_{i,k}, S_{k,j} \rangle$, it first fetches the two lists from appropriate processors, with associated context switches, and then processes them locally. The local computation is a doubly nested loop that, for every pair $\langle b, c \rangle \in S_{i,k} \times S_{k,j}$, looks up production

rules whose right hand side is b and c, generates symbols included in $S_{i,j}$, and updates $E_{i,j}$. The ratio between the local computation and the communication + switch overhead determines an upper bound of the achievable speed-up.

Critical Path: This is relevant for short sentences. We denote the process that operates on the pair $\langle S_{i,k}, S_{k,j} \rangle$ as $P_{i,k,j}$ (That is, calculation of $S_{i,j}$ consists of $(j-i-1)$ processes $\{P_{i,k,j}\}$ $(i < k < j)$). Since $P_{i,k,j}$ depends on the result of $S_{i,k}$ and $S_{k,j}$, there can be no overlap between $P_{i,k,j}$ and computation of $S_{i,k}$ or $S_{k,j}$. In other words, there cannot be any overlap between $P_{i,k,j}$ and any of $P_{i,*,k}$ or $P_{k,*,j}$.

Let us write $P_{i,k,j} > P_{i',k',j'}$ to denote that there cannot be overlap between $P_{i,k,j}$ and $P_{i',k',j'}$. Thus, we have:

$$\begin{aligned} & P_{i,k,j} > P_{i,x,k} \quad (i < x < k), \\ \text{and} \quad & P_{i,k,j} > P_{k,x,j} \quad (k < x < j) \end{aligned}$$

Using this fact recursively, we have a chain of $P_{i,k,j}$s each of which can never overlap. The length of a chain can be as long as the length of the input sentence minus one. For example,

$$P_{0,1,n} > P_{1,2,n} > P_{2,3,n} > \cdots > P_{n-2,n-1,n}.$$

Load Imbalance: Critical path is not a relevant limiting factor for long sentences. Even so, processor utilization is limited by load imbalance. Our current mapping of threads onto processors seems not adequate in several ways. First, the amount of work performed by each thread is not constant and depends on input. As a general hint, threads for long sub-sentences tend to perform large tasks, but an accurate estimation of the work performed by threads seems very difficult. Second, even when sentences are very long (say, have 100 words), the number of threads we have is not sufficiently large to make the round-robin distribution very effective. As a result, we consistently observe fairly large idle time in Chapter 6, even for longest sentences on small number of processors.

In summary, CKY imposes severe implementation challenges. When we look at results in Chapter 6, where processors spent approximately the same

amount of time on busy, overhead, and idle, overhead and load imbalance are both significant limiting factors. Assuming we would eliminate all these idle times, the overhead alone would still limit the achievable speed-up on p processors to something around $0.5p$. Assuming the overhead would become very close to zero, the idle time caused by load imbalance alone would still limit the achievable speed-up on p processors to something around $0.5p$. Improving load balance requires us generate more threads and distribute them on processors in a smaller unit, but this in turn makes the overhead limit severer. Similarly, shortening the critical path by making threads finer raises the overhead.

A.3 RNA

A.3.1 Problem

The problem really is to predict feasible secondary structures of a given RNA sequence. Here we omit all such biological aspects and describe the problem only from computational point of view.

Let us begin with some preliminary definitions. A *stack region* is designated by its *position* and *energy*. A position is represented by a quadruple of integers, though details are unimportant. Energy is specified by a positive floating point number. We assume there is a binary relation that determines if two stack regions are *compatible*. A set of stack regions is *feasible* if any pair of its elements is compatible. The energy of a set of stack regions is the total energy of all the elements. Here is the problem.

> Given a set of stack regions S and a parameter Δ, find all feasible subsets of S that have an energy no smaller than $(E_{\max} - \Delta)$, where $E_{\max}$ is the optimal energy over any feasible subsets of S.

Typical problem sizes (the number of stack regions in S) we tested are between 170 and 230. The maximum size we have so far solved on 256 CPUs is 280.

A.3.2 Basic Algorithm

Since the problem is to find *all* feasible subsets that have certain energies, we must use a combinatorial brute-force search algorithm, rather than heuristics

to obtain an approximation. We can, of course, do better than the naive algorithm that tests 2^n possible combinations. As preprocess, we divide S into disjoint partitions, each partition of which does *not* have any pair of compatible stack regions. Each partition is called *incompatible islet* [27]. This preprocessing takes $O(n^2)$. We also augment each incompatible islet with the maximum energy among its elements. We call it the *achievable energy* of the islet.

Once S is partitioned into incompatible islets, a subset of S is specified by selecting one or zero element from each incompatible islet, because, by the definition of the incompatible islet, we cannot choose two or more stack regions from any single incompatible islet. We form the search tree along this view. That is, let $S = I_1 + \cdots + I_m$ where I_k is an incompatible islet. The root node has $(\#I_1 + 1)$ direct child nodes, $\#I_1$ nodes of which indicate cases where an element has been committed from I_1 and the last child node the case where none has been selected from I_1. Similarly, each direct child of the root has $(\#I_2 + 1)$ children. As a heuristics, we sort islets by its size, from smaller ones to larger ones. This makes pruning described later more effective. Assuming depths at which pruning occur are approximately constant regardless how islets are ordered, expanding smaller number of branches near the root of a search tree tends to produce less work in total.

How do we prune unnecessary branches? We keep track of the maximum energy achieved so far and prune a node that can never achieve that value under the node. Incompatible islet plays an important role to estimate the value achievable under a given node of the search tree. The estimated value for a search node is the total energy of already committed stack regions + the total achievable energies over the incompatible islets yet examined. This value is actually computed incrementally; the estimated value for the root is the total achievable energies over all the incompatible islets. Whenever we go down the tree, committing one region or none from an islet, we subtract the difference between the achievable energy of the islet and the energy actually committed.

A.3.3 Description in ABCL/f and Parallelization

Parallelization is really simple. We simply express the search algorithm using recursion and extract parallelism by recursive future calls. Load distribution

is simply done by making calls remote until a certain threshold depth and then proceeding locally. Load balancing is achieved simply by making much larger number of remote calls than the number of processors and distributing them randomly. In the experiments we specified the threshold value in the command line.[1] We typically fork from 60K to 200K threads by remote future calls.

All processors share the maximum energy so far achieved (M) and keep them consistent. Since M monotonically increases and there are no problems about underestimating it, the strict consistency is of course unnecessary. Processors merely have to update M promptly enough to make pruning effective. A processor broadcasts the value when it achieves a value of energy that is significantly better than M at any intermediate nodes of the search tree, or when it achieves a value of energy that is better than M at leaves of the search tree. The threshold value that determines exactly when a processor updates M in intermediate nodes is chosen arbitrarily by the user.

A.3.4 Behavior and Performance Limiting Factors

Since so many threads are randomly distributed on processors and threads never synchronize until termination, each processor has plenty of parallelism, any of which can be scheduled at anytime. This makes this application very latency-tolerant, as we have observed in Chapter 3.

Two potential limiting factors are load imbalance and overhead. As we have observed in Chapter 6, when the problem size is large, each accounts for less than 10% of the execution time on any number of processors. To summarize, as far as this problem is concerned, the current simple approach that chops the work into small pieces and distributes them randomly is successful. Further improvements are of course possible by more sophisticated load balancing method that minimizes the number of remote fork, while achieving a similar or a better load balance.

[1] Determining an appropriate value adaptively is not actually difficult. Given thread creation is inexpensive, we do not have to "pinpoint" the right threshold. We merely have to guarantee that the number of threads are not too small. We let the user to specify simply for benchmarking purpose.

Appendix B

Unboxed Channel Scheme

This chapter details a code generation scheme that implements the unboxed channel scheme outlined in Section 5.4, using a simple language that models ABCL/f. The model language has all the essential features in ABCL/f, including variable bindings, assignments, conditionals, operations on channels, and asynchronous procedure invocations. Heap locations are not explicitly included, but, for our purpose, sufficiently modeled by channels. We first define the syntax of the language and then develop a source to source transformation scheme that inserts explicit boxing operations wherever necessary. The transformed expression guarantees that channels are never escaped in their unboxed representation to places where we cannot maintain the semantics of first class channels.

B.1 Essential Syntax

We define the essential syntax of the language as follows.

E	=	c	(constant)
	|	`$`	(new channel)
	|	v	(variable)
	|	$v := E$	(assignment)
	|	`let` $v = E$ `in` E	(let)
	|	`if` E `then` E `else` E	(conditional)
	|	E `<=` E	(reply)
	|	$*E$	(touch)
	|	$op\ (E, E)$	(primitives)
	|	$E\ (E)$ `@` E	(invocation)

c is a constant, which we assume is not a channel. `$` is a special constructor that creates a new empty channel. v is a variable. A variable is introduced either by a let, a regular parameter of a procedure, or a reply channel of a procedure. $v := e$ assigns the value of e to v. The value of an assignment is unit. `let` $v = e$ `in` e' binds e to v and evaluates e' under the extended environment. The value of the let expression is the value of e'. e `<=` e' is the reply expression. It puts the value of e' to the value e, which is a channel. The value of this expression is unit. $*e$ is a touch expression. It extracts a value from the value of e, which is a channel. The value of a touch expression is the extracted value. $op\ (e, e)$ is a primitive operation, which we assume takes two parameters and returns a value that is not a channel. We further assume a primitive operation does not allocate channels or write values into channels. $e\ (e')$ `@` e'' is the canonical procedure invocation expression. It invokes procedure e, passing a regular parameter e' and a reply channel e''. The value of this expression is the value of e''.

The syntax of a procedure definition is:

$$v\ (v')\ \texttt{@}\ v'' = E,$$

where v is the name of the procedure, v' the name of the regular parameter, v'' the name of the reply channel, and E the body expression that is evaluated when this procedure is invoked. For example, a procedure that calls two procedures `f` and `g` in parallel is written as follows.

```
f_and_g (x) @ r = let s = f (x) @ $
                  in
                    let t = g (x) @ $
                    in
                      r <= *s + *t
```

That is, two channels are created and procedures f and g are called with the created channels as the reply channels. The new channels are bound to variable r and s and finally touched at the last line. In examples that follow, we use a let expression to bind multiple variables sequentially and to have multiple expressions in the body. It is a syntactical abbreviation of nested let expressions.[1]

We note differences between ABCL/f and the simple language developed. First, ABCL/f has mutable records that encode concurrent objects and deftype data. For the purpose of developing code generation schemes for unboxed channels, any heap data can be modeled by channels. Specifically, a record creation can be modeled by a channel creation + putting the elements into the channel. Reading a value from a record is modeled by getting a value from a channel and writing a value to a channel by putting a value to a channel. This captures all the aspects we are currently interested in—where and how to insert boxing operations. Second, ABCL/f has block expression of Common Lisp, whose value is the value of its last expression or any expression specified by return-from expression in the block. For example, a block expression:

```
(block L
  (if ...
      (return-from L x)
      ...)
  y)
```

returns x as soon as `(return-from L x)` is evaluated. Otherwise, it returns y when control reaches the last line. For our purpose, the only important aspect of a block expression is that multiple expressions can become the value of the entire expression. Thus it is sufficient to model a block expression as

[1] The scope rule of the extended let binding is that of ML's `let` val bindings, or Common Lisp's `let*` bindings. That is, the scope of a variable is the body and all bound expressions that follow the binding.

a (possibly nested) if expression whose branches correspond to all possible return expressions of a block.

B.2 Locations

Below, we use term "*locations*" to mean variables introduced by let, regular parameters of procedures, reply channel parameters of procedures, and channels. The first three categories are just called "*variables*". Variables introduced by let or regular parameters are called "regular variables."

We say a value is *bound* to a location when the location holds the reference to the value. We say a value channel is *bound* if it is bound to at least one location. Assignment, let, touch, reply, and procedure invocation change the binding between locations and values.

B.3 Boxing

Where the compiler inserts a boxing operation is represented by three translation functions B_0, B_{+1}, and B_∞. Each function takes an expression and returns another expression in which boxing operation is explicitly performed by `box` operator. The operand of a `box` operator is either a channel creation expression (`$`) or a variable.

$B_u(c)$	=	c
$B_u(\$)$	=	`box $` $(u = \infty)$
	=	`$` (otherwise)
$B_u(v)$	=	`box` v ($u = +1$ or ∞ and v is either a let-variable or the reply channel of a procedure)
	=	v (otherwise)
$B_u(v := e)$	=	$v := B_{+1}(e)$
$B_u($`let` $v = e$ `in` $e')$	=	`let` $v = B_{+1}(e)$ `in` $B_u(e')$
$B_u($`if` e `then` e' `else` $e'')$	=	`if` $B_0(e)$ `then` $B_\infty(e')$ `else` $B_\infty(e'')$
$B_u(e$ `<=` $e')$	=	$B_0(e)$ `<=` $B_\infty(e')$
$B_u(*e)$	=	$*B_0(e)$
$B_u(op\ (e, e'))$	=	$op\ (B_\infty(e), B_\infty(e'))$
$B_u(e\ (e')$ `@` $e'')$	=	$B_0(e)\ (B_{+1}(e'))$ `@` $B_u(e'')$
$B_D(v\ (v')$ `@` $v'' = e)$	=	$v\ (v')$ `@` $v'' = B_0(e)$

The above definition of B_u actually defines three functions B_0, B_{+1}, and B_∞ simultaneously. Each of them takes an expression and returns an augmented expression. B_D transforms a procedure definition. The returned expression or definition is identical to the original, except that some variables and channel creations (`$`) are wrapped by the boxing operator box. When x is a channel, `box` x allocates a heap channel according to the state of x and returns the boxed channel as the value of the expression. Furthermore, if x is a variable, box x updates x by the pointer to the boxed channel. When x is not a channel, it is nop. In statically typed monomorphic languages like ABCL/f, whether an expression is a channel or not is checked statically. In the following examples, we omit box operations where the operand is obviously not a channel.

An expression returned by B_u guarantees the following two properties.

- An unboxed channel is never bound to multiple regular variables, a regular variable + a reply parameter, or channels.
- For any sub-expression that may be evaluated to a bound unboxed channel, the compiler can find a variable that (must) reference it.
-

The first condition says that the set of locations that binds an unboxed channel include at most one regular variable and (possibly multiple) reply channel parameters. We require the second condition to guarantee that operations on a bound unboxed channel can correctly update the variable that references it.

Let us illustrate the augmentation using a simple example.

```
f_and_g (x) @ r = let s = f (x) @ $
                      t = g (x) @ $
                  in
                    r <= *s + *t
```

The augmentation of `f (x) @ $` proceeds as follows:

$$\begin{aligned} B_{+1}(\texttt{f (x) @ \$}) &= B_0(\texttt{f})\ (B_{+1}(\texttt{x}))\ \texttt{@}\ B_{+1}(\texttt{\$}) \\ &= \texttt{f (box x) @ \$} \end{aligned}$$

Thus, the entire procedure definition is augmented as follows.


```
f_and_g (x) @ r = let s = f (box x) @ $
                      t = g (box x) @ $
                  in
                    r <= *s + *t
```

The regular argument to `f` and `g`, which is `x`, is boxed because it may be a channel that is used after these invocations. If `x` were passed in its unboxed form, the sharing relationship between `f`, `g`, and the caller would not be properly maintained. The reply channels of these invocations, `$`, are not boxed, on the other hand. The protocol described in Section 5.4 guarantees that `f` and `g` return the updated representation of the reply channel to the caller. If, for example, `f` escapes its reply channel to heap, the reply channel is boxed there and the boxed representation is returned back to the caller.

Consider another example that demonstrates a channel may be boxed because we otherwise violate the second condition. For example,

```
f_or_g (p) @ r = let s = f (0) @ $
                     t = g (1) @ $
                 in
                   r <= *(if p then s else t)
```

is augmented as follows.

```
f_or_g (p) @ r = let s = f (0) @ $
                     t = g (1) @ $
                 in
                   r <= *(if p then (box s) else (box t))
```

In this example, `s` (`t`) is boxed despite it is the only location that references the channel. This is because otherwise the surrounding `*` expression would not know which channel it should touch, and thus it does not know which variable should it update in place.

B.4 Correctness

Since we have not developed a formal semantics of the model language, we cannot formally prove the correctness of the above augmentation. In this section, we argue the correctness of the above augmentation under the

informal semantics described in Section B.1 and the following assumptions about code generation:

- Boxing operation on a variable, box v where v is a variable, updates v in place and affects any subsequent references to v. This is most easily maintained by mapping a variable in the model language to a single location in the generated code (*e.g.*, a single C variable) and implementing box v by updating the variable.
- On a procedure invocation, the callee is scheduled first, and, when control returns back to the caller, the callee passes the reply channel to the caller. The reply channel is boxed when the callee's local variables still reference the reply channel. If a caller's local variable references the reply channel after the invocation, the caller updates the variable by the value passed by the callee before proceeding.
- In effect, the protocol introduced in the second item 'postpones' the boxing operation, which would normally have to be done as soon as a channel is bound to the reply channel parameter of the callee. It defers the boxing operation until the caller is scheduled again. In other words, a reply channel can remain unboxed as long as it is no longer referenced by the callee *when control returns to the caller*.

Let E_∞, E_{+1}, and E_0, be the sets of expressions that are produced by B_∞, B_{+1}, and B_0, respectively. From the above augmentation rules, it follows that E_∞, E_{+1}, and E_0 are generated by the following syntax:

$$
\begin{aligned}
E_\infty \;=\;& c \\
\mid\;& \texttt{box \$} \\
\mid\;& \texttt{box } v \\
\mid\;& v := E_{+1} \\
\mid\;& \texttt{let } v = E_{+1} \texttt{ in } E_\infty \\
\mid\;& \texttt{if } E_0 \texttt{ then } E_\infty \texttt{ else } E_\infty \\
\mid\;& E_0 \texttt{ <= } E_\infty \\
\mid\;& \texttt{*}E_0 \\
\mid\;& op\ (E_\infty, E_\infty) \\
\mid\;& E_0\ (E_{+1})\ \texttt{@}\ E_\infty
\end{aligned}
$$

$$
\begin{aligned}
E_{+1} \;=\;& c \\
\mid\;& \texttt{\$} \\
\mid\;& \texttt{box } v \\
\mid\;& v := E_{+1} \\
\mid\;& \texttt{let } v = E_{+1} \texttt{ in } E_{+1} \\
\mid\;& \texttt{if } E_0 \texttt{ then } E_\infty \texttt{ else } E_\infty \\
\mid\;& E_0 \texttt{ <= } E_\infty \\
\mid\;& \texttt{*}E_0 \\
\mid\;& op\ (E_\infty, E_\infty) \\
\mid\;& E_0\ (E_{+1})\ \texttt{@}\ E_{+1}
\end{aligned}
$$

$$
\begin{aligned}
E_0 \;=\;& c \\
\mid\;& \texttt{\$} \\
\mid\;& v \\
\mid\;& v := E_{+1} \\
\mid\;& \texttt{let } v = E_{+1} \texttt{ in } E_0 \\
\mid\;& \texttt{if } E_0 \texttt{ then } E_\infty \texttt{ else } E_\infty \\
\mid\;& E_0 \texttt{ <= } E_\infty \\
\mid\;& \texttt{*}E_0 \\
\mid\;& op\ (E_\infty, E_\infty) \\
\mid\;& E_0\ (E_{+1})\ \texttt{@}\ E_0
\end{aligned}
$$

From the above syntax, we observe the following properties about E_∞, E_{+1}, and E_0.

- E_∞ is never evaluated to an unboxed channel.
- When E_{+1} is evaluated to an unboxed channel, the value is not bound to any location at moment the expression has been evaluated.
- When E_0 is evaluated to an unboxed channel, the locations that bind the value include at most one variable at moment the expression has been evaluated.

Except for procedure invocations in E_{+1} and E_0, claims are easily verified by checking how E_∞, E_{+1}, and E_0 are constructed (a formal proof would require an induction on the structure of expressions, which we omit because

our discussion is already informal). The claim about procedure invocations is subtler and requires a precise definition of "the moment the expression has been evaluated." From the protocol assumed above, a procedure invocation first transfers control to the callee, which eventually resumes to the caller either by blocking or termination. We define a procedure invocation has been evaluated when the control returns to the caller, whether the callee terminated or not.

Suppose that an (augmented) procedure invocation $f\ (x)\ \mathtt{@}\ r$ ($\in E_{+1} \cup E_0$) has been evaluated to an unboxed channel. The protocol introduced above guarantees that, at moment it has been evaluated, the callee's local variables no longer bind the value of r. Furthermore, the value of r is not bound to channels, because if it were, the callee must have evaluated an expression c `<=` x, where x is evaluated to the channel. The argumentation rules guarantee that $x \in E_\infty$, which implies that the reply channel would have been boxed. To sum up, at moment $f\ (x)\ \mathtt{@}\ r$ has been evaluated to an unboxed channel, there are no references to the value of r either in the callee or channels. Now let us verify claims about E_{+1} and E0 in turn.

When $f\ (x)\ \mathtt{@}\ r \in E_{+1}$, $r \in E_{+1}$ also hold. By the inductive hypothesis, we have that the value of r is not bound to any location at moment r has been evaluated. This implies that when $f\ (x)\ \mathtt{@}\ r \in E_{+1}$ has been evaluated, the value of r is not referenced by the caller. Since the callee or channels do not reference the value of r either, it is not bound to any location at all. Similarly, when $f\ (x)\ \mathtt{@}\ r \in E_0$, $r \in E_0$ also hold. By the inductive hypothesis, we have that the value of r is bound to at most one variable at moment r has been evaluated. This implies that when $f\ (x)\ \mathtt{@}\ r \in E_0$ has been evaluated, the value of r is bound to at most one variable in the caller. Since the callee or channels do not reference the value of r either, it is referenced by at most one variable.

Having shown the claims about E_∞, E_{+1}, and E_0, we have that:

- Whenever a channel is stored into channel, it is boxed.
- Whenever a channel is bound to a regular variable, either by an assignment, a let binding, or a procedure invocation, the channel is either boxed or not referenced from any other location.
- Whenever a channel is bound to a reply parameter, it is either boxed or bound to at most one local variable.

What remains to be shown is how to generate code from `*`e or e `<=` e' where e may be evaluated to an unboxed channel. Specifically, we must clarify how to determine the variable to update? The following function V takes an expression in E_0 that may be evaluated to an unboxed channel referenced from a variable. It returns an empty set if the value is not bound to any variable or returns the variable that refers to the value, if it is bound to a variable.

$$\begin{aligned} V(\mathtt{\$}) &= \{\} \\ V(v) &= \{v\} \\ V(\mathtt{let}\ v = e\ \mathtt{in}\ e') &= V(e') \\ V(e\ (e')\ \mathtt{@}\ e'') &= V(e'') \end{aligned}$$

When the code generator emits the code for `*`e or e `<=` e' where e is a channel, the generated code checks if $V(e)$ is still unboxed and if it is, updates it. Note that a conditional expression is never evaluated to an unboxed channel, so we can precisely determine the variable to update, which *must* reference the channel we are operating.

分散記憶並列計算機のための効率的で再利用可能な細粒度マルチスレッディング及びゴミ集め

田浦健次朗