An Efficient Load-Balancing Framework Based on Lazy Partitioning of Sequential Programs

Masahiro Yasugi[1], Tsuneyasu Komiya[2], and Taiichi Yuasa[1]

[1] Graduate School of Informatics, Kyoto University
[2] Information and Computer Sciences, Toyohashi University of Technology

Abstract. This paper proposes an efficient load-balancing framework based on lazy partitioning of sequential programs where a single-threaded program is basically executed with little parallelization overhead but its computation is divisible for efficient utilization of multiple computing resources. Traditional fork-join multithreaded languages can be implemented on the top of our framework, but our framework provides an intermediate abstraction layer where a new thread is created as a task only when dynamic partitioning is requested at runtime. To divide a running task by spawning a new task, the sequential program can have "spawn-request handlers", which are similar to "exception handlers", but they are "called" rather than used for non-local exits. The oldest available spawn-request handler is called to obtain good load balancing, because it tends to spawn a task with sufficient amount of work and to minimize the number of task creations and transfers. The sequential program can also have "undo-redo clauses", which are similar to Java's `finally` clauses, but an undo-redo clause can perform necessary state recovery (undo/backtracking) before an attempt to call an older spawn-request handler and perform the redo (the cancellation of the undo) after the attempt. By using undo-redo clauses, we can avoid undesirable copying of temporarily-modified data (e.g., for backtrack search problems). To implement the proposed constructs, our translator into an extended C language uses "nested functions" provided by the GNU C compiler. We also enhanced GCC to reduce allocation overhead and maintenance overhead of nested functions. The results of preliminary performance measurements on various shared-memory parallel computers exhibit near-ideal speedups and quite low parallelization overhead.

1 Introduction

Recently we can often utilize multiple computing resources. For example, we can use computers in the Internet, processors in a parallel computer, and simultaneous multi-threading units in a processor (such as Hyper-threading). For efficient utilization of multiple computing resources, each computing resource should always have exactly one task: both poor parallelism (a lower utilization rate) and excessive parallelism (higher managing overhead) should be avoided.

To realize efficient dynamic load balancing by transferring tasks among computing resources in fine-grained parallel computing such as search problems, load

balancing schemes which lazily create and extract a task by splitting the present running task, such as *Lazy Task Creation* (LTC)[1], are effective. Here we use the term "task" to refer to a schedulable unit of work which is also transferable to other computing resources. In the case of multithreaded languages, a task may internally contain and manage multiple language-level threads. By extracting a task around the *root* of the invocation tree as much as possible, each computing resource can have sufficient amount of work with the almost minimum number of task transfers.

The rest of this paper is organized as follows. We discuss multithreaded languages in Sect. 2. In Sect. 3, we propose a load-balancing framework based on lazy partitioning of sequential programs where a single-threaded program is basically executed with little parallelization overhead but its computation is divisible. Traditional fork-join multithreaded languages can be implemented on the top of our framework, but we can employ our framework directly if we prefer the speed rather than the higher abstraction. Section 4 proposes the implementation of our framework: we show that we can generate an extended C program with LTC-based load balancing where callers' variables are accessed by using *nested functions* provided by the GNU C compiler[2, 3]. We also show that we can perform *backtracking* by using nested functions. Here "backtracking" means not only to backtrack to a point where a new choice for the search can be made but also to undo the side effect of the previous examined choices as in a sequential backtrack search. In addition, we propose a semantical separation of nested functions from normal top-level functions, which enables a significant performance improvement. Section 5 shows the measured performance on shared-memory parallel computers. The results exhibit near-ideal speedups and quite low parallelization overhead.

2 Multithreaded Languages and Parallel Execution

High-level programming languages for parallel processing are quite useful to develop reliable, reusable and efficient applications on various parallel architecture including shared-memory architecture and distributed-memory architecture. In multithreaded languages, every (mostly) independent computation can be naturally expressed as a thread. There are various multithreaded languages which are based on sequential languages and have additional constructs for multithreading: Multilisp[4] is a `future`-based multithreaded Scheme language, Cilk[5] is a multithreaded C language and OPA[6, 7] is a multithreaded Java language. In these languages, parallel programs are written with threads, expecting that the language systems provide automatic load balancing.

In multithreaded languages, threads can be allocated to available computing resources. However, multithreaded programs cost parallelization overhead compared to sequential (single-threaded) programs. To reduce the parallelization overhead, multiple threads allocated to a particular computing resource should statically be fused into a single thread. But the static allocation of threads for efficient utilization of heterogeneous (and dynamic) computing resources is quite

```
{
  int x = spawn fib(n-2);
  int y = fib(n-1);
 join:
  z = x + y;
}
```

```
{
  int s = 0;
  for(i=0;i<n;i++)
    if(test(i))
      s += spawn search(i);
 join:
  z = s;
}
```

Fig. 1. Fork-join multithreaded code examples.

difficult. In addition, the amount of work each thread has is not always statically predictable. Thus the static partitioning of parallel programs has only limited efficiency, and the dynamic load balancing with dynamic (re)allocation of present threads is required.

2.1 Scheduling and Load Balancing

A multithreaded program incurs the overhead of thread creation/termination and that of synchronization, even when there are enough number of threads to utilize available computing resources. Indeed, an important goal of various sophisticated implementation techniques of multithreaded languages is the removal of the overhead (e.g., by using specialized code). One of the best implementation techniques of multithreaded languages is LTC, which is originally proposed for Multilisp's `future` construct. In LTC, a newly created thread is directly and immediately executed like a usual call while (the *continuation* of) the oldest thread in the computing resource may be stolen by other idle computing resources. Usually, the idle computing resource (*thief*) randomly selects another computing resource (*victim*) for stealing a task.

Fork-join style multithreaded computation is an important class of parallelism. Let us focus on that style below. Figure 1 illustrates fork-join multithreaded code examples in multithreaded C. The left one is based on OPA[7]; the first function call is executed by a newly created *child* thread and the *parent* thread can execute the second function call in parallel, then after the join of the child thread it uses the results. The right one is based on Cilk[5]; the parent thread repeatedly creates a child thread if the condition is satisfied, then it waits for the join of all created child threads. If a parent thread never has to wait for some conditions other than the join of child threads (this is true in many cases), suspending the parent thread at any time to schedule child threads is permitted; i.e., the parent thread can *call* a child thread sequentially.

There are roughly two types of scheduling strategies for efficient multithreaded execution: (1) Child-first: at fork point, the computing resource executes the child thread prior to the parent thread and makes the parent thread stealable for other computing resources, and resumes the parent thread if it has not been stolen after the completion of the child thread. (2) Parent-first: at fork point, the computing resource executes the parent thread prior to the child thread and

makes the child thread stealable for other computing resources, and calls the child thread if it has not been stolen at the join point of the parent thread. Cilk[8, 5, 9], LTC[1, 10], StackThreads/MP[11] and OPA[7] belong to the former category, and WorkCrews[12], Lazy RPC[13], FJTask[14, 15] and Leapfrogging[16] belong to the latter category. Note that some of these[12, 14, 15] are multithreading frameworks rather than multithreaded languages, and some of these [1, 10, 11, 7, 16] are used in more general multithreading parallelism than the well-structured fork-join parallelism.

There are some variations of the child-first scheduling strategy. Instead of stealing the whole parent thread, a thief can steal the parent thread's *partial continuation*[17–19] before the join point. For example, in the right program of Fig. 1, when the parent thread spawns `search(i)`, the partial continuation of the parent thread is identical to the execution of:

"`i++; for(;i<n;i++) if(test(i)) s += spawn search(i);`".

A thief can also steal only a child thread which will be created during the execution of the partial continuation. For example, if `test(n-1)` is true, the thief can steal the child thread for `search(n-1)`. These two variations of scheduling strategies can be realized on the top of our load-balancing framework. Our framework also supports the parent-first strategy.

Another subtle issue is what a computing resource should do when the current thread is waiting for the join of the previously *stolen task* (which is the child thread or the partial continuation of the parent thread) running on the another computing resource. If the computing resource steals a task from a randomly selected victim and calls[1] it to hide the waiting latency, the maximum stack size might be unexpectedly large[16]. As was stated in [13], this may not be a serious problem, but a strategy to guarantee the maximum stack size is proposed in Leapfrogging[16], where the computing resource waiting for the join of the stolen task steals back a task from the computing resource that previously stole the stolen task. We employs this idea of Leapfrogging in our framework.

2.2 Locality and Sharing Problems

In traditional multithreaded languages, each thread must use its own working space to avoid interference between threads. This degrades reference locality by increasing the working set size. For example, on 1GHz Pentium-III, a sequential program which calculates a 16-dimensional vector $\Sigma_{i=1}^{1000000}$`inpvec[`i`]` by using a single accumulation vector runs about 12% faster than by using some of `inpvec[`i`]` as accumulation vectors to avoid interference between threads. This does not change the number of operations but simply changes the working spaces for multiple threads; i.e., reference locality has an effect on the performance. If we do not want to destroy the value of `inpvec[`i`]`, intermediate accumulation vectors should be allocated and merging of intermediate results is necessary. In

[1] Since we limit the expressed parallelism to the fork-join style, the transfered task can be executed with the stack of the suspended task[12–15].

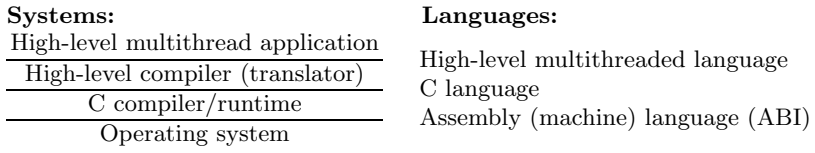

Fig. 2. Abstraction layers with C compiler.

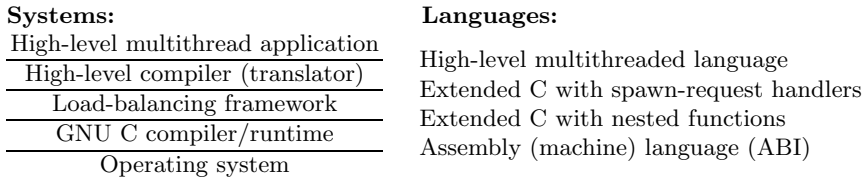

Fig. 3. Abstraction layers with load-balancing framework.

this case, the sequential program using a single accumulation vector runs about 29% faster than using intermediate accumulation vectors.

In traditional multithreaded languages, threads may share a data structure such as an array if the data structure is "read only", but they cannot share the array if the array is modified by threads even if modification is performed only temporarily. So, undesirable copying of temporarily-modified data is necessary. In sequential programs, copying of temporarily-modified data can be avoided if the subsequent subprogram observes the recovered state (e.g., for backtrack search problems). This technique promotes further reuse/sharing of the working space in sequential programs. For example, on 1GHz Pentium-III, a sequential program which finds all possible solutions of the Pentomino puzzle by reusing a single data structure for the current configuration runs about 104% faster than copying the data structure. Note that the "real" modification can be avoided by using a thread-local "virtual" modification history, but the access time to such an array will no longer be a constant but proportional to the history size.

3 Load-Balancing Framework Based on Lazy Partitioning

Multithreaded languages can be implemented on the top of assembly languages as in [1]. However, an appropriate intermediate language and its system is desirable for portability. In Cilk[5] and OPA[7] implementations, the C language is used as an intermediate language as in Fig. 2. The high-level compilers have to provide the functionality of load balancing since the C compiler does not provide it.

We propose an efficient load-balancing framework based on lazy partitioning of sequential programs where a single-threaded program is basically executed with little parallelization overhead but its computation is divisible for efficient utilization of multiple computing resources. Traditional fork-join multithreaded languages can be implemented on the top of our framework, but our framework

provides an intermediate abstraction layer for load-balancing (Fig. 3). In addition, we can employ our framework directly if we prefer the speed rather than the higher abstraction. We will compare our intermediate-level framework language with the high-level multithreaded languages.

In this section, we present the interface and functionality of our load-balancing framework. The implementation of our framework will be presented in Sect. 4. In our framework, a new thread is created as a task only when dynamic partitioning is requested at runtime.

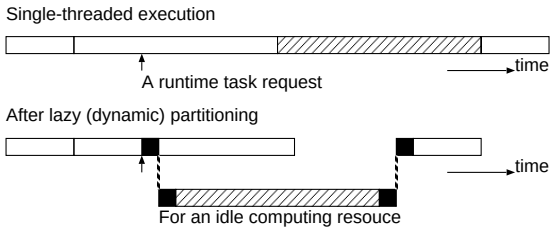

Fig. 4. Lazy partitioning of sequential programs.

Figure 4 illustrates the single-threaded execution and the parallel execution after the lazy (dynamic) partitioning. The sequential execution enjoys smaller working set size and reuse/sharing of the working space just before the partitioning; nevertheless the parallel execution after the division is the same as that of multithreaded languages. To realize this partitioning, our language has "spawn-request handlers" and "undo-redo clauses".

To divide a running task by spawning a new task, a sequential program can have "spawn-request handlers", syntactically denoted by

"`do {`*body*`} handle_spawn_req(`*req*`){`*spawn-attempt*`}`",

which are similar to "exception handlers" but are "called" rather than used for non-local exits. When a runtime task request is received during the execution of *body*, the oldest available spawn-request handler is called to obtain good load balancing, because it tends to spawn a task with sufficient amount of work and to minimize the number of task creations and transfers. If the oldest spawn-request handler does not spawn a task, the second oldest spawn-request handler is invoked; this chain of handler invocations is repeated until a task is spawned or all spawn-request handlers are invoked. For example, the two spawn-request handlers in Fig. 5 are invoked in the displayed order.

In traditional multithreaded languages, each thread must use its own working space to avoid interference between threads. Since the spawn-request handler or the spawned task can assign a working space on demand, our framework improves reference locality by reducing the working set size. and obtains better performance.

```
do{
  do{
     may receive spawn request (0.)
  }handle_spawn_req(req){ called if task has not been spawned (2.)  }
}handle_spawn_req(req){   called (1.) }
```

Fig. 5. The invocation order of spawn-request handlers.

```
do{
  do{
    do{
      do{
         may receive spawn request (0.)
      }undo_redo{
         undo (1.)
          try_to_spawn;
         redo (6.)
      }
    }handle_spawn_req(req){  called if task has not been spawned (5.) }
  }undo_redo{
     undo (2.)
     try_to_spawn;
     redo (4.)
  }
}handle_spawn_req(req){  called (3.) }
```

Fig. 6. The invocation order of undo-redo clauses.

The sequential program can also have "undo-redo clauses", syntactically denoted by

"`do {`*body*`} undo_redo {`*undo* `try_to_spawn;` *redo*`}`",

which are similar to Java's `try-finally` or Scheme's `dynamic-wind`: an undo-redo clause can perform necessary state recovery (undo/backtracking) before an attempt to call an older spawn-request handler and perform the redo (the cancellation of the undo) after the attempt. For example, the spawn-request handler and the undo-redo clauses in Fig. 6 are invoked in the displayed order. By using undo-redo clauses, we can avoid undesirable copying of temporarily-modified data (e.g., for backtrack search problems) and promote further reuse/sharing of the working space.

The spawn-request handlers and undo-redo clauses are dynamically scoped like exception handlers.

3.1 Examples

We will introduce two examples of tree-recursive fine-grained parallel computing; these applications are used to explain the details of our proposal in the following sections. These are also used for performance measurements.

The first example is to recursively compute the n-th term of Fibonacci sequence defined as follows:

$$\mathrm{fib}(1) = \mathrm{fib}(2) = 1$$

$$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \text{ for } n > 2$$

Of course, this computation has no practical meaning, but this is a simple tree-recursive irregular computation and is very useful for measurements of overhead (e.g., overhead of function calls and/or overhead of parallelization). Since each function has few actual work, the measured overhead well represents the worst case overhead for similar tree-recursive computations.

```
int fib(int n)
{
  if (n <= 2)
    return 1;
  else
    {
      int s = 0;
      int x = fib(n-2) !! { s += fib(n-1); }
      return s + x;
    }
}
```

Fig. 7. Multithreaded Fibonacci computations.

Figure 7 presents multithreaded Fibonacci computations based on Lazy RPC[13]. This high-level code can be translated onto our framework as in Fig. 8. The program is basically a single-threaded program except for the spawn-request handler which specifies the way how to spawn (`SET_TASK` and `SEND_TASK`) a new thread/task by accessing variables `xt` and `n`. Variable `UNSETP(xt)` indicates whether or not the computation has proceeded to the second recursive call. The second recursive call is usually called sequentially, but it may become a new thread/task to respond a task request sent from some idle computing resource. In this program, a task using `int` value and producing `int` value is explicitly described. Such information on the stolen task can be kept in `xt`; thus, it is possible to recover from the fault of the computing resource that stole the task if the fault can be detected at "`WAIT_RESULT(xt)`".

The second example is a search problem to find all possible solutions of the Pentomino puzzle. A pentomino consists of five equal-sized squares attached edge-to-edge to form some shape. There are twelve possible pentominos that can be formed in this way. The Pentomino puzzle is to fill the 6×10 rectangular board with the twelve pentominos. Of course, this problem is only a puzzle, but it represents many similar search problems.

All possible solutions of this puzzle can be computed by "backtrack search". backtrack search can be viewed as search through a tree of partial solutions, called *backtrack search tree*, where the root represents an empty solution, the children of a tree node represent partial solutions extended by possible one-step choices, and a leaf represents either a solution or a partial solution with no further extension. In sequential backtrack search, the search tree is traversed in a depth-first manner. Typically, only one partial solution is kept in memory and

```
int fib(int n)
{
  if (n <= 2)
    return 1;
  else
    {
      int s = 0;
      int x;
      TASK_DATA(int,int) xt = INIT_TASK_DATA(int, int);
      do {
        s += fib(n-1);
      } handle_spawn_req(req) {
        if(UNSETP(xt))
          {
            SET_TASK(xt, fib, n-2);
            SEND_TASK(req, xt);
          }
      }
      if (UNSETP(xt))
        x = fib(n-2);
      else
        {
          WAIT_RESULT(xt);
          x = xt.result;
        }
      return s + x;
    }
}
```

Fig. 8. Fibonacci computations with lazy partitioning.

one-step extension updates the partial solution; backtracking must undo the side effect of the previous examined choices. In the Pentomino puzzle, a piece is put at the next available position by one-step extension and the piece is removed by backtracking.

However, this "in-place" update/undo assumes that a single processor solely performs these operations; it is difficult to apply the same scheme to parallel backtrack search. Thus, when an partial solution is extended for parallel backtrack search, the extended partial solution must be kept in different memory to keep the original partial solution unchanged. This incurs a significant copying overhead.

Figure 9 shows a *backtrack search* algorithm for the Pentomino puzzle using in-place updating/restoring of partial solutions. The call to a spawn-request handler also involves the backtracking (and undoing side effects, i.e., removing the pieces for the Pentomino example) along the search tree. In traditional multithreaded languages, each partial solution of the backtrack search tree must always be copied for unsharing them among multiple threads. In the proposed language, copying is necessary only when dynamic partitioning is performed.

```
private int a[12];          // piece
private int b[70];          // board

int try_piece(int k, int i){
  variables decls
  for(d=0;d<n;d++){
     if (!  room available?) goto Ln;
      put the piece
     do {
       find the first empty location in kk
       s += search(kk);     // recursive search
     } undo_redo {
       remove the piece     // backtrack
       try_to_spawn;
       put the piece        // cancel backtrack
     }
      remove the piece      // backtrack
    Ln:
     continue;
  }
  return s;
}

int search(int k){
  variables decls
  if(found) return 1;
  if(prunable) return 0;
  while (i<si) {
     int ii = i++;
     if (a[ii] == 0)
       do {
         s += try_piece(k, ii);
       } handle_spawn_req(req) {
         adjust si, copy a and b, and spawn try_piece(k,si)
       }
  }
  while (i<12) {
     int ii = i++;
     WAIT_RESULT(r[ii]);
     s += r[ii].result;
  }
  return s;
}
```

Fig. 9. Lazy partitioning with an undo-redo clause.

4 Implementation

This section proposes the implementation techniques of our load-balancing framework. As was shown in Fig. 3, our load-balancing framework is implemented on the top of an extended C language with nested functions which is provided by the GNU C compiler.

4.1 An Extended C Language with Nested Functions

Compilers (translators) for multithreaded languages generate low-level code. In the original LTC[1], assembly code is generated to directly manipulate the execution stack. Both translators for Cilk[5] and OPA[7] generate C code. Since it is illegal and not portable for C code to directly access the execution stack, the Cilk/OPA translators generate two versions (fast/slow) of code; the fast version code saves values of live variables in a heap-allocated frame upon call (in the case of Cilk) or return (in the case of OPA) so that the slow version code can continue the rest of computation based on the heap-allocated saved *continuation*.

In the conventional C language, once a processor calls a function, the callee cannot (directly) access the caller's local variables. They are sleeping until the return to the caller's function. Thus the processor cannot (without pointers, which interfere many compiler optimization techniques) refer to the important and fundamental information the original call-sites have.

Assembly languages are very powerful but machine-dependent, while C language is (almost) machine-independent but lacks an ability to access the variables slept in the execution stack. This problem motivates researchers to develop new powerful machine-independent intermediate languages, such as C-- [20, 21]. C-- has an ability to access the variables slept in the execution stack by using "stack walking" although its primary goal seems to be the language-level support of exception handling.

Recently, we found that nested functions can be used for a processor to legally refer to the contents (i.e., variables' values) of its execution stack. This is useful in parallel processing, since multiple processors can interact with each other based on the fundamental and important information in the execution stacks. Furthermore, each processor can change its prospective behavior after several returns by modifying the values in its execution stack. the GNU C Compiler (GCC)[3] provides such nested functions [2] as an extension to C. We expect that the GCC's extended C with nested functions can also be used as a powerful machine-independent intermediate language.

In GCC, a "nested function" is a function defined within another (nested or top-level) function. Nested function definitions are permitted in the places where variable definitions are allowed; that is, in any block, before the first statement in the block. The name of the nested function is local to the block. A nested function can access the lexically-scoped variables in the allocation-time environment and its pointer can be used as a function pointer to indirectly call the closure (that is, a pair of the nested function and its environment). For example, in Fig. 10, when `h` indirectly calls the function `g`, it can access parameter `x` and local variable `x`

```
int f(int x) {
  int y = x * x;
  int g(int z) { return x + y + z; }
  return h(g, 0);
}
```

Fig. 10. Nested functions.

slept in `f`'s frame. A unique closure is (logically) created every time the control arrives at the definition of the nested function in the same way as allocations of `auto` variables or `auto` arrays. Since a closure is stack-allocated, the pointer to a closure cannot be used after the exit of the block where the nested function is defined.

4.2 Lazy Backtracking Task Creation: a Translator into an Extended C Language

To implement the proposed constructs, the programs on our load balancing framework are translated into an extended C language with nested functions. We call the implementation/translation techniques *Lazy Backtracking Task Creation* (LBTC). LBTC has the following features:

- LBTC is based on message passing[10]; that is, employing a polling method.
- LBTC only supports well-structured fork-join parallelism and a task is created to process a (sub)computation in fork-join style parallel computations as in WorkCrews[12], Lazy RPC[13] and FJTask[14, 15].
- When a (sub)computation is not stolen at the join point, the computation is executed in an inlined manner.
- LBTC uses no (de)queue of tasks or computations: when a processor is requested of a task, it calls a nested function which calls an older nested function in a nested manner to extract a task nearest to the stack bottom. The lack of task queues indicates that LBTC has remarkable laziness.
- When a processor become idle with an empty stack, it randomly chooses a *victim* processor to steal a task and sends a task request and waits for a reply.
- When a (sub)computation has been stolen but not yet finished, the processor tries to steal *back* a task from the processor that has stolen the (sub)computation. In this way, the maximum stack size can be guaranteed as in Leapfrogging[16].

A task steal proceeds as follows (see Fig. 11): (A) When a *victim* processor receives a task request (Fig. 11:1), it backtracks by calling nested functions in a nested manner and it creates and transfers a task (Fig. 11:2), and it cancels the backtracking to resume computation. When it creates a task, it also modifies the caller's variable so that it will wait for the result of the stolen task (Fig. 11:3).

By translating the Fibonacci program shown in Fig. 8, we obtain the main part of the Fibonacci program shown in Fig. 12. "`xt_unsetp = 1`" means that

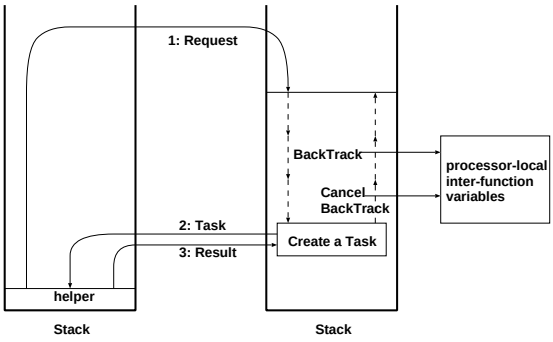

Fig. 11. Proposed task steal protocol.

there is one thread candidate for stealing (by other processors) or inlining (by the current processor). `bk` is the nested function and its address is passed upon calls. First, `bk` calls the older nested function passed via parameter `bkf`, then tries to create a task if `bkf` or deeper nested functions have not made a task yet (if `ret` is 0). `POLL` checks if a request is available and, if so, starts calling the nested function. If the parent computation reaches the join point and notices the child computation has been stolen, it tries to steal *back* a task from the processor which has stolen the child computation.

For the parallel backtrack search, extracting a task also involves the backtracking (and undoing side effects, i.e., removing the pieces for the Pentomino example) along the search tree. The compilation of undo-redo clauses is straightforward; the clause is implemented by a nested function and `try_to_spawn;` simply calls the older nested function.

4.3 An Extended C Compiler

We propose a semantical separation of nested functions from normal top-level functions, which enables a significant performance improvement.

GCC implements taking the address of a nested function using a technique called "trampolines"[2]. Trampolines are code fragments generated on the stack at runtime to indirectly enter the nested function with a necessary environment. The runtime code generation is performed for the allocation of the nested function. If we can distinguish a nested function from normal top-level functions, we can implement its *closure* with a stack-allocated pair of pointers for the nested function and the environment. By using a stack-allocated pointer pair instead of a dynamically-generated code fragment, the allocation overhead of nested functions can be reduced.

We enhanced GCC-3.2 (with a patch of 700 lines) to involve new entities called *closures* other than functions to eliminate "trampolines". This can be implemented by extending the syntax with a new keyword `closure` and modifying the RTL generation phase[3].

```
int fib(proc_env pr,
        int (*bkf)(req_buf_t *), int n){
  int ret = 1;
  int s, x;
  void **saved_req_port_p;
  int result_buf;
  int result_port;
  int xt_unsetp;
  int bk(req_buf_t *req_buf_p){
    /* a nested call of the older nested function */
    if(ret) ret = bkf(req_buf_p);
    if(ret) return 1;  /* a task has been created at an older caller */
    if(xt_unsetp){
      task_buf_t *tbp =req_buf_p->task_buf_p;
      int *tpp        =req_buf_p->task_port_p;
      xt_unsetp = 0;
      /* save info for "request back" */
      saved_req_port_p =req_buf_p->req_port_p;
      /* result not available yet */
      result_port = 0;
      /* put a task */
      tbp->f = tf_fib;
      tbp->sender = pr->myid;
      tbp->a.t1.result_buf_p = &result_buf;
      tbp->a.t1.result_port_p = &result_port;
      tbp->a.t1.n = n-2;
      /* transfer the task */
      finish_write_before_write();
      atomic_write_int(*tpp, 1);
      return 1;
    }
    return 0;
  }
  if(n <= 2) return 1;
  s = 0;
  xt_unsetp = 1;
  POLL(pr->req_port, bk);
  s += fib(pr, bk, n-1);
  if(xt_unsetp){
    x = fib(pr, bkf, n-2);
  }else{
    /* wait for thread completion */
    while(atomic_read_int(result_port) == 0)
      /* request back a task and run*/
      steal_run_task(pr,saved_req_port_p,1);
    start_read_after_read();
    x = result_buf;
  }
  return s + x;
}
```

Fig. 12. compiled code for Fibonacci.

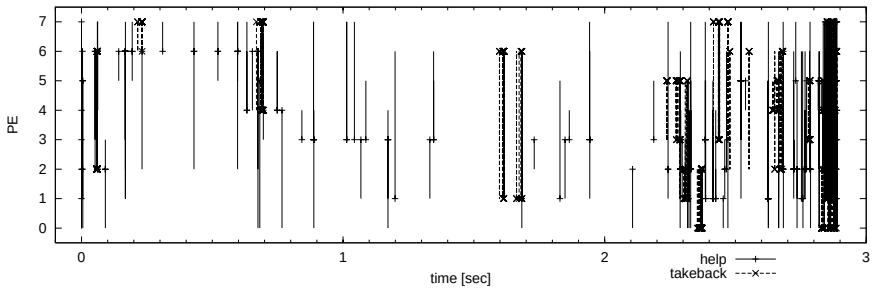

Fig. 13. Record of task transfers by the Pentomino program on 8 processors.

We also enhanced GCC-3.2 (with a patch of 1500 lines) to involve new entities called *L-closures* other than closures to eliminate maintenance overhead of nested functions, which enables variables shared among different nesting-level functions to get callee-save registers as long as the L-closures is not invoked. This can be implemented by extending the syntax, modifying the RTL generation phase and prologue/epilogue generation phase, and adding small runtime assembly code. The details of the implementation of L-closures will be reported by a separate paper.

5 Preliminary Performance Measurements

We collected the results of performance measurements with fib(36) on various shared-memory parallel computers:

- Sun Ultra Enterprise 10000 (250MHz UltraSPARC-II, 1MB L2 Cache, 64CPUs) Solaris 7, gcc 2.95.2 `-O2 -mcpu=ultrasparc`
- IBM RS/6000 SP (332MHz PowerPC 604e instruction 32KB/data 32KB L1 cache, 0.25MB L2 cache, 4CPUs) AIX Version 4.3, gcc 2.95.2 `-O2 -mcpu=powerpc`
- PC (1GHz Pentium-III, 2CPUs), Solaris 8, gcc 2.8.1 `-O2`

We also collected the results of performance measurements with the Pentomino application on the shared-memory parallel computers.

First, all applications achieved near-ideal speedups; for example, 30.9 times speedup with 32 CPUs and 45.4 times speedup with 48 CPUs were obtained with the Pentomino application on Ultra Enterprise 10000, (we think that some dropping was caused by pruning.) and nearer-ideal speedups were obtained for all other cases. Figure 13 shows the task transfers between 8 processors with the Pentomino program. The horizontal axis is time and the vertical axis is for processor numbers. `help` represents the case where an idle processor with an empty stack successfully steals a task. `takeback` represents the case where an waiting processor with a non-empty stack successfully steals *back* a task. We

can see the number of task transfers is moderate while good load balancing is obtained.

Next, we can focus on the overhead of parallelization. The elapsed time of a sequential C program and that of the parallel program on a single PE are shown in Table 1 and Table 2. The relative time to the sequential C program is shown in parenthesis. Since Fibonacci programs include few actual computation, the overhead of parallelization are emphasized. We think the main overhead is for allocation of nested functions with "trampolines". SPARC needs to flush some instruction cache for runtime-generated trampolines code. Note that PowerPC-based AIX does not use "trampolines"; so the overhead is smaller than other machine configurations. In Pentomino programs, we think the main overhead is for maintenance of nested functions. Because nested functions cannot directly access the register values of the enclosing functions, the enclosing functions need to save shared values in the stack. Since Pentium has only a small number of registers, the maintenance overhead is smaller than RISC processors.

Table 1. Single PE execution time compared to sequential C (Fibonacci).

		(sec)
	C	LBTC
SPARC	2.36	7.88 (3.33)
PowerPC	1.94	3.52 (1.81)
Pentium	0.537	1.20 (2.24)

Table 2. Single PE execution time compared to sequential C (Pentomino).

		(sec)
	C	LBTC
SPARC	14.4	23.0 (1.60)
PowerPC	9.41	12.4 (1.32)
Pentium	3.76	4.61 (1.23)

5.1 Performance Enhancement and Comparison

In the following, the results on the SPARC are measured on 750MHz Ultra-SPARC-III. Table 3 and Table 4 show the results of GCC enhancement for eliminating "trampolines" by using "closures" and "L-closures". The tables also present the measured performance of Cilk 5.3.2[8].

In the case of SPARC, we obtained significant overhead reduction by using "closures"; that is, Fibonacci's overhead of 334% is reduced to 61%. Note that this overhead includes not only allocations (plus maintenance) of nested

functions but also polling checks of task requests. By using "L-closures", the overhead is reduced to 20%. The results show that our framework with "closures" or "L-closures" is fairly faster than Cilk's multithread implementation, and the speed of our framework using "L-closures" is almost comparable to the sequential execution.

By using undo-redo clauses, we can avoid copying overhead of temporarily-modified data. On Pentium, the "closure" program for Pentomino using undo-redo clauses runs about 87% faster than using copying. In Table 4, "Cilk1" incurs the copying overhead. "Cilk2" also incurs copying overhead between parent and child threads, but it partly reuses the recovered temporarily-modified data among sibling threads if the parent ensures all child threads are joined, which can be inspected via `SYNCHED` pseudo variable[8].

Table 3. Performance Enhancement (Fibonacci).

(sec)

	C	LBTC	closure	L-closure	Cilk
SPARC	0.80	3.47 (4.34)	1.29 (1.61)	0.96 (1.20)	3.27 (4.08)
Pentium	0.50	1.07 (2.16)	0.95 (1.91)	0.80 (1.62)	2.52 (5.09)

Table 4. Performance Enhancement (Pentomino).

(sec)

	C	LBTC	closure	L-closure	Cilk1	Cilk2
SPARC	4.47	8.09 (1.81)	6.58 (1.47)	4.85 (1.09)	18.2 (4.08)	8.96 (2.01)
Pentium	3.68	4.36 (1.19)	4.28 (1.16)	3.90 (1.06)	9.94 (2.70)	7.19 (1.95)

6 Discussion

As was shown in Fig. 3, our load-balancing framework provides lower abstraction layer than traditional multithreaded languages. For example, our program in Fig. 8 gives more details of its execution than multithreaded program in Fig. 7, but the description of our program can be troublesome. So, if the translation from high-level multithreaded languages is possible, it will be preferable in most cases and the role of our framework is simply the support of the multithreaded language implementation.

In some cases, however, it may be preferable employing our framework directly:

– Since a single-threaded program is executed basically, debugging (tracing execution and monitoring spawn status) is easier than multithreaded languages.

– Since the spawn-request handler can package all necessary data in the new task, load balancing among *distributed* computers can easily be supported. By contrast, shared-memory is usually assumed in multithreaded languages. Moreover, by saving the packages at victims, it is possible to recover from the faults of thieves.
– To parallelize a sequential program, a working space for each thread must be considered in multithreaded languages. By contrast, Consideration on the interference between the present task and spawned task is enough for lazy partitioning. Our framework improves reference locality by reducing the working set size. Moreover, by using undo-redo clauses, we can avoid undesirable copying of temporarily-modified data.

7 Conclusion

This paper proposes an efficient load-balancing framework based on lazy partitioning of sequential programs where a single-threaded program is basically executed with little parallelization overhead but its computation is divisible for efficient utilization of multiple computing resources.

Our implementation of the dynamic load balancing framework, *lazy backtracking task creation* (LBTC), uses nested functions to backtrack to a point where a task for other computing resources can be created and to temporarily undo the side effect of the running calls as in a sequential backtrack search.

Since LBTC has remarkable laziness, its performance is very close to that of the usual sequential programs, except for the overhead of allocating nested functions. The results of performance measurements on various shared-memory parallel computers were shown in this paper, exhibiting near-ideal speedups and quite low parallelization overhead.

The nested functions are provided by the GCC using a technique called "trampolines"[2]. We also enhanced GCC to eliminate "trampolines"; this reduced the overhead of allocating nested functions significantly and we obtained less than 92% parallelization overhead which include allocation and maintenance of nested functions and polling of task requests. We also enhanced GCC to reduce the maintenance overhead of nested functions. As a result, we obtained less than 63% parallelization overhead in the cases including the case where the Cilk 5.3.2 implementation incurs about 400% parallelization overhead.

References

1. Mohr, E., Kranz, D.A., Halstead, Jr., R.H.: Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems **2** (1991) 264–280
2. Breuel, T.M.: Lexical closures for C++. In: Usenix Proceedings, C++ Conference. (1988)
3. Stallman, R.M.: Using the GNU Compiler Collection. Free Software Foundation, Inc. for gcc-3.2 edn. (2002)

4. Halstead, Jr., R.H.: New ideas in parallel Lisp: Language design, implementation, and programming tools. In Ito, T., Halstead, R.H., eds.: Parallel Lisp: Languages and Systems. Volume 441 of Lecture Notes in Computer Science., Sendai, Japan, June 5–8, Springer, Berlin (1990) 2–57
5. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. ACM SIGPLAN Notices (PLDI'98) **33** (1998) 212–223
6. Yasugi, M.: Hierarchically structured synchronization and exception handling in parallel languages using dynamic scope. In: Proc. of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications. (1999)
7. Umatani, S., Yasugi, M., Komiya, T., Yuasa, T.: Pursuing laziness for efficient implementation of modern multithreaded languages. In: Proc. of the 5th International Symposium on High Performance Computing. Number 2858 in Lecture Notes in Computer Science (2003)
8. Supercomputing Technologies Group: Cilk 5.3.2 Reference Manual. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA. (2001)
9. Strumpen, V.: Indolent closure creation. Technical Report MIT-LCS-TM-580, MIT (1998)
10. Feeley, M.: A message passing implementation of lazy task creation. In: Proceedings of International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications. Number 748 in Lecture Notes in Computer Science, Springer-Verlag (1993) 94–107
11. Taura, K., Tabata, K., Yonezawa, A.: StackThreads/MP: Integrating futures into calling standards. In: Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP). (1999) 60–71
12. Vandevoorde, M.T., Roberts, E.S.: WorkCrews: An abstraction for controlling parallelism. International Journal of Parallel Programming **17** (1988) 347–366
13. Feeley, M.: 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, Springer-Verlag (1995) 3–21
14. Lea, D.: A Java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande, ACM Press (2000) 36–43
15. Lea, D.: Concurrent Programming in Java: Design Principles and Patterns. Second edn. Addison Wesley (1999)
16. Wagner, D.B., Calder, B.G.: Leapfrogging: A portable technique for implementing efficient futures. In: Proceedings of Principles and Practice of Parallel Programming (PPoPP'93). (1993) 208–217
17. Felleisen, M.: The theory and practice of first-class prompts. In: 15th Annual ACM Symposium on Principles of Programming Languages (POPL '88). (1988) 180–190
18. Queinnec, C., Serpette, B.: A dynamic extent control operator for partial continuations. In: 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91). (1991) 174–184
19. Moreau, L., Queinnec, C.: Partial continuations as the difference of continuations: A duumvirate of control operators. In: 6th Programming Language Implementation and Logic Programming (PLILP '94). Volume 844 of Lecture Notes in Computer Science. (1994) 182–197
20. Jones, S.P., Ramsey, N., Reig, F.: C--: a portable assembly language that supports garbage collection. In: International Conference on Principles and Practice of Declarative Programming. (1999)

21. Ramsey, N., Reig, F.: A single intermediate language that supports multiple implementations of exceptions. In: Proc. of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI2000). (2000) 285–298