Lightweight Lexical Closures for Legitimate Execution Stack Access

Masahiro Yasugi, Tasuku Hiraishi, and Taiichi Yuasa

Graduate School of Informatics, Kyoto University, Japan
{yasugi, hiraisi, yuasa}@kuis.kyoto-u.ac.jp

Abstract. We propose a new language concept called "L-closures" for a running program to legitimately inspect/modify the contents of its execution stack. L-closures are lightweight lexical closures created by evaluating nested function definitions. A lexical closure can access the lexically-scoped variables in the creation-time environment and indirect calls to it provide legitimate stack access. By using an intermediate language extended with L-closures in high-level compilers, high-level services such as garbage collection, check-pointing, multithreading and load balancing can be implemented elegantly and efficiently. Each variable accessed by an L-closure uses private and shared locations for giving the private location a chance to get a register. Operations to keep coherency with shared locations as well as operations to initialize L-closures are delayed until an L-closure is actually invoked. Because most high-level services create L-closures very frequently but call them infrequently (e.g., to scan roots in garbage collection), the total overhead can be reduced significantly. Since the GNU C compiler provides nested functions, we enhanced GCC at relatively low implementation costs. The results of performance measurements exhibit quite low costs of creating and maintaining L-closures.

1 Introduction

Implementing sophisticated machine code generators for a variety of platforms is not easy work. Therefore, many compiler writers for high-level languages use C as an almost portable and machine-independent intermediate language; that is, they write only translators from high-level languages into C.

Most compiled C programs use execution stacks for efficiency. Upon a function call, a stack frame is allocated not only for parameters and local variables of the function but also for the return address, the previous frame pointer, the callee-save registers and `alloca`-ed spaces. Efficient support for some high-level run-time services (such as garbage collection, self-debugging, stack-tracing, check-pointing, migration, continuations, multi-threading and/or load balancing) requires inspecting/modifying the contents of execution stacks. In C, however, once a function is called, the callee cannot efficiently access the caller's local variables. Some local variables may have the values in callee-save registers, and *pointer*-based accesses interfere with many compiler optimization techniques. In addition, the stack frame layout is machine-dependent and direct stack manipulation by the running C program via forged pointers is illegal in essence, because

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 170–184, 2006.

the data of stack frames are not application-level data (values) but *meta-level* data for execution. Illegal access will also open security issues.

For example, to implement garbage collection (GC), the collector needs to be able to find all *roots*, each of which holds a reference to an object in the garbage-collected heap. In C, a caller's pointer variable may hold an object reference, but it may be sleeping in the execution stack. Even when using direct stack manipulation, it is difficult for the collector to distinguish *roots* from other elements in the stack. *Stack maps* may be used, but they are not inherent C data and need special compiler support. For this reason, conservative collectors[1] are usually used with some limitations. When a *copying collector* is used to implement GC, it needs to be able to accurately scan all *roots* since the objects are moved between semi-spaces and all root pointers should refer to the new locations of objects. Accurate copying collection can be performed by using translation techniques based on "structure and pointer"[2, 3], but translating local variables into structure fields invalidates many compiler optimization techniques.

This problem motivates researchers to develop new powerful and portable intermediate languages, such as C--[4, 5]. C-- is a portable assembly language (lower-level than C) but it has the ability to access the variables sleeping in the execution stack by using the C-- runtime system to perform "stack walk". Thus, C-- can be used as an intermediate language to implement high-level services such as garbage collection.

This paper proposes yet another intermediate language, which is an extended C language with a new language concept called "L-closures" for a running program to legitimately inspect/modify the contents of its execution stack (i.e., the values of data structures and variables). L-closures are lightweight lexical closures created by evaluating nested function definitions. A lexical closure can access the lexically-scoped variables in the creation-time environment and indirect calls to it provide legitimate stack access. Compared to C--, our approach more elegantly supports high-level services, and needs quite low implementation costs by reusing the existing compiler modules and related tools such as linkers.

The rest of this paper is organized as follows: Section 2 presents our motivating example. In Sect. 3, we show the design of the proposed language features (*closures* and *L-closures*), where we propose a semantical separation of nested functions from ordinary top-level functions. Section 4 proposes our implementation model for L-closures. Section 5 presents our current implementation based on GCC. The results of performance measurement are discussed in Sect. 6. The results exhibit quite low costs of creating and maintaining L-closures. Section 7 discusses the costs and applications of L-closures together with the related work, and shows that many high-level services can be implemented by translating into the extended C language.

2 A Motivating Example

Let us consider a high-level program which recursively traverses binary tree nodes and creates an associative list with the corresponding search data. Such a

```
Alist *bin2list(Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  if(x->right) rest = bin2list(x->right, rest);
  kv = getmem(&KVpair_d);              /* allocation */
  kv->key = x->key; kv->val = x->val;
  a = getmem(&Alist_d);                /* allocation */
  a->kv = kv; a->cdr = rest;
  rest = a;
  if(x->left) rest = bin2list(x->left, rest);
  return rest;
}
```

Fig. 1. A motivating example: tree-to-list conversion

```
typedef void *(*move_f)(void *);

                /* scan0 is an L-closure pointer. */
Alist *bin2list(lightweight void (*scan0)(move_f),
                Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  /* scan1 is an L-closure, and pass it on the following calls. */
  lightweight void scan1(move_f mv){  /* nested function */
    x = mv(x); rest = mv(rest);       /* roots scans */
    a = mv(a); kv = mv(kv);           /* roots scans */
    scan0();                          /* for older roots */
  }
  if(x->right) rest = bin2list(scan1, x->right, rest);
  kv = getmem(scan1, &KVpair_d);    /* allocation */
  kv->key = x->key; kv->val = x->val;
  a = getmem(scan1, &Alist_d);      /* allocation */
  a->kv = kv; a->cdr = rest;
  rest = a;
  if(x->left) rest = bin2list(scan1, x->left, rest);
  return rest;
}
```

Fig. 2. Scanning GC roots with L-closures (nested functions)

high-level program may be translated into a C program shown in Fig. 1. Here, `getmem` allocates a new object in heap, and a copying collector needs to be able to scan all *root* variables such as `x`, `rest`, `a` and `kv` even when `bin2list` is being recursively called.

In the proposed intermediate language, a program with copying GC can be elegantly expressed as in Fig. 2. Allocator `getmem` may invoke the copying collector with L-closure `scan1` created by evaluating the nested function definition. The copying collector can indirectly call `scan1` which performs the movement (copy) of objects using roots (`x`, `rest`, `a` and `kv`) and indirectly calls L-closure `scan0` in a nested manner.[1] The actual entity of `scan0` may be another instance of `scan1` in the caller. By repeatedly invoking L-closures until the bottom of the stack is reached, all roots in the entire execution stack can be scanned.

[1] Alternatively, `scan1` may return `scan0` to eliminate tail calls.

In Fig. 2, `bin2list`'s variables (`x`, `rest`, `a` and `kv`) should have chances to get (callee-save) registers. However, if we employ the typical Pascal-style implementation for L-closures, `bin2list` must perform memory operations (much slower than register operations) to access these variables because `scan1` also accesses the values of these variables in the stack memory usually via a static chain. Note that the same problem arises in translation techniques for stack-walking based on "structure and pointer"[2, 3].

Our goal is to reduce these costs of *maintaining* L-closures (i.e., to enable register allocation) by using a new implementation policy for L-closures. The policy also reduces the costs of *creating* L-closures but accepts higher invocation costs. Because most high-level services create L-closures very frequently but call them infrequently (e.g., to scan roots in garbage collection), the total overhead can be reduced significantly.

3 Design

Pascal and many modern programming languages other than C (such as Lisp, Smalltalk, and ML) permits a function defined within another (nested or top-level) function. We employ Pascal-style nested functions for our extended C language. It can access the lexically-scoped variables in the creation-time environment and a pointer to it can be used as a function pointer to indirectly call the lexical closure (that is, a pair of the nested function and its environment). A lexical closure is (logically) created every time the control arrives at the nested function definition in the same way as local variable definitions. Since a closure is created on the stack, unlike garbage-collected languages, the pointer to a closure cannot be used after the exit of the block where the nested function is defined.

We propose a semantical separation of nested functions from ordinary top-level functions, which enables a significant performance improvement by using different calling sequences for nested functions. For this purpose, we introduce a language concept called **closures**, which have almost the same roles as ordinary (top-level) functions but which are not regarded as ordinary functions. We extend the language syntax with a keyword `closure` in the same way as keyword `lightweight` in Fig. 2. A program which passes a closure pointer as an ordinary function pointer will produce a type error, and vice versa.[2]

We also introduce a new language concept called **L-closures** other than ordinary functions and other than *closures*. We extend the syntax with a keyword `lightweight` as in Fig. 2. That is, there are two types of lexical closures, and each type has the following goals and limitations:

Closures are intended to employ the Pascal-style implementation (i.e., static chains). *Closures* do not keep interoperability with ordinary top-level functions. The owner function of *closures* involves substantial costs of maintaining *closures*. *Closures* have moderate creation/invocation costs. These costs are the same as the corresponding costs for techniques based on "structure and pointer."

[2] In practice, coercing function pointers into closure pointers may be permitted.

L-closures are intended to employ the implementation policy for aggressively minimizing costs of creating and maintaining L-closures by accepting higher invocation costs. L-closures do not keep interoperability with ordinary top-level functions (and *closures*). L-closures are callable only from within the owner function and its descendants in the caller-callee relation. (e.g., not callable from different threads)

We can choose an adequate type according to a situation. For example, L-closures should be used to implement most of the high-level services discussed in Sect. 2 and Sect. 7, because those L-closures are rarely called and minimizing the creation/maintenance costs is desired.

4 Implementation Models

In this section, we propose recommended implementation models for closures and L-closures.

We can implement a *closure* with a stack-allocated pair of pointers. The pointer pair consists of the actual nested function and the environment (static chain). The *closure* pointer can refer to the pair. When a caller indirectly calls a closure, it already distinguish the closure pointer from ordinary function pointers by compile-time looking at the type of the pointer, then it loads the static chain (the second element of the pointer pair) into the static chain register and calls the actual nested function (the first element of the pointer pair). Note that we cannot use the pointer pair directly as a two-word closure pointer, since C permits interoperability between the generic `void *` type and any other pointer type.

To minimize costs of creating L-closures, the initialization of an L-closure is delayed until it is actually called. This means that the creation cost of L-closures is virtually zero (similar to carefully-implemented exception handlers.)

To minimize costs of maintaining L-closures, if a function f has a nested function g of L-closure type and g accesses f's local variable (or parameter) x, x uses *two* locations, namely a *private* location and a *shared* location, for giving the private location a chance to get a (callee-save) register by reducing interference with the existing optimizers and register allocators. Note that x does not use a private location if the address of x is taken, or if a nested function accessing x is not of L-closure type. In addition, if g has a nested function g_2, g_2's access to f's variable is accounted to be g's access regardless of g_2's type.

The similar technique (but incomplete in terms of lazy pre/post-processing) can be expressed in extended C with nested functions as in Fig. 3 for the function `bin2list` in Fig. 2. The function `bin2list`, which owns a nested function `scan1`, introduces private variables (such as `p_x` and `p_a`) and uses the private ones except for the function calls. Upon a function call, `bin2list` saves the private values into the shared variables (such as `x` and `a`) as *pre-processing*. When the control is returned, it restores the private values from the shared variables as *post-processing*. This technique gives the private variable a chance to get a register. However, this technique cannot omit pre/post-processing even if `scan1`

```
Alist *bin2list(void (*scan0)(move_f), Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  /* scan1 is a lexical closure, and pass it on the following calls. */
  void scan1(move_f mv){              /* nested function */
    x = mv(x); rest = mv(rest);       /* roots scans */
    a = mv(a); kv = mv(kv);           /* roots scans */
    scan0();                          /* for older roots */
  }
  /* private variables */
  Bintree *p_x = x, Alist *p_rest = rest, *p_a = a; KVpair *p_kv = kv;
  if(p_x->right){
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    Alist *_r = bin2list(scan1, p_x->right, p_rest);
    p_x = x, p_rest = rest, p_a = a, p_kv = kv,  /* post-processing */
    p_rest = _r;
  }
  {
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    KVpair *_r = getmem(scan1, &KVpair_d);       /* allocation */
    p_x = x, p_rest = rest, p_a = a, p_kv = kv;  /* post-processing */
    p_kv = _r;
  }
  p_kv->key = p_x->key; p_kv->val = p_x->val;
  {
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    Alist *_r = getmem(scan1, &Alist_d);         /* allocation */
    p_x = x, p_rest = rest, p_a = a, p_kv = kv;  /* post-processing */
    p_a = _r;
  }
  p_a->kv = p_kv; p_a->cdr = p_rest;
  p_rest = p_a;
  if(p_x->left){
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    Alist *_r = bin2list(scan1, p_x->left, p_rest);
    p_x = x, p_rest = rest, p_a = a, p_kv = kv,  /* post-processing */
    p_rest = _r;
  }
  return p_rest;
}
```

Fig. 3. Adding private variables and pre-processing and post-processing in C. This is not the real code but shown for explanation purpose only.

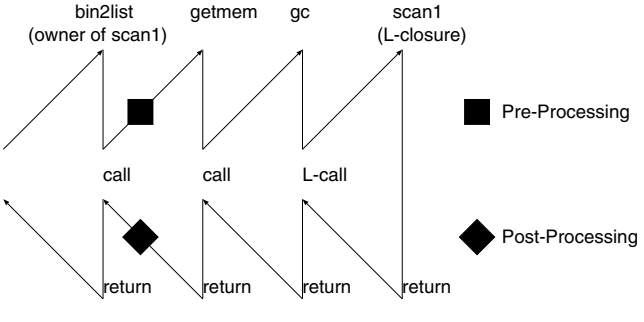

Fig. 4. Usual pre-processing and post-processing

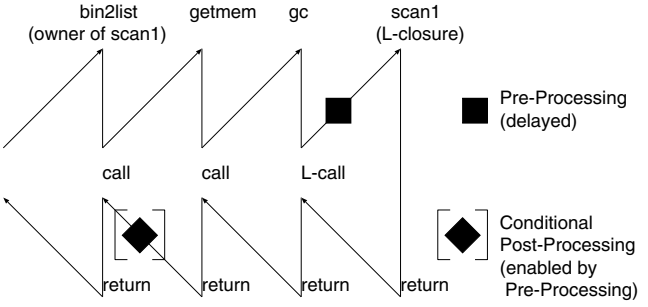

Fig. 5. Delayed pre-processing and conditional post-processing performed only if the L-closure is actually called

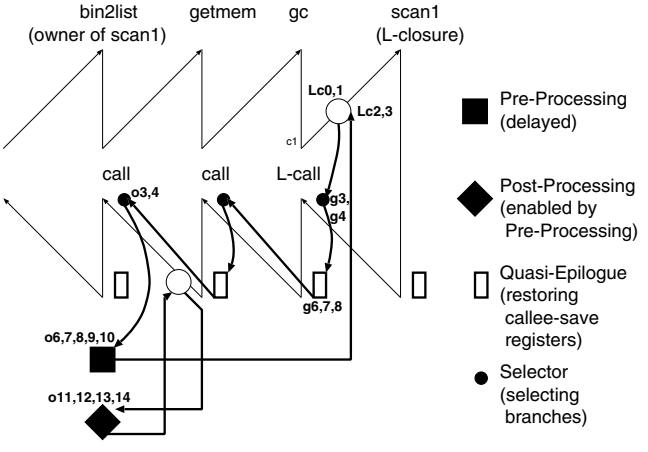

Fig. 6. (Non-local) temporary return to the owner of the L-closure to be called for correct pre-processing. Annotated numbers correspond to those in Fig 7.

is not actually called. The control flow on pre-processing and post-processing at the time when `scan1` is being called by `gc` can be depicted as in Fig. 4.

To overcome this problem, we propose *delayed pre-processing* and *conditional post-processing* as in Fig. 5. The pre-processing is delayed until the call to the L-closure, and the conditional post-processing is dynamically enabled by pre-processing. *Pre-processing* (indicated by filled squares) consists of the following steps: (1) initializing all L-closures (function-pointers and static chains), (2) copying private values into shared locations, and (3) enabling post-processing (by changing return addresses). Performing pre-processing more than once[3] is avoided (i.e., pre-processing is also conditional) by checking if the conditional post-processing is already enabled. *Post-processing* (indicated by filled diamonds) simply copies values from shared locations into private locations.

[3] In the case of recursive calls of an L-closure.

```
bin2list: // owner of scan1
o0  : ...
o1  : call getmem with selector o3.
o2  : ...
o3  : /* selector for o1 */
o4  : if (L-closure to be called is in this frame) jump to pre-processing o6.
o5  : else jump to quasi-epilogue o18.
o6  : /* pre-processing for o1 */
o7  : copy values from private locations to shared locations.
o8  : initialize all L-closures (function-pointers and static chains).
o9  : save and modify o1's return address to enable post-processing o11.
o10 : continue the L-call according to on-stack info.
o11 : /* post-processing for o1 */
o12 : save the return value.
o13 : copy values from shared locations to private locations.
o14 : continue the actual return.
o15 : ...
o16 : /* selector for modified return addresses */
o17 : continue the L-call according to on-stack info.
o18 : /* quasi-epilogue */
o19 : restore callee-save registers.
o20 : temp-return to the selector for the previous frame.

gc:  // caller of scan1 (= scan)
g0  : ...
g1  : L-call scan with selector g3.
g2  : ...
g3  : /* selector for g1 */
g4  : jump to quasi-epilogue g6.
g5  : ...
g6  : /* quasi-epilogue */
g7  : restore callee-save registers.
g8  : temp-return to the selector for the previous frame.

L-call f:
Lc0 : save f and registers.
Lc1 : temp-return to the selector for the previous frame.
Lc2 : restore f and registers.
Lc3 : setup static chain for f and jump to f.
```

Fig. 7. pseudo code and calling steps

Since we give each private location a chance to get a callee-save register, restoring callee-save registers (including the frame pointer but excluding the stack pointer) before pre-processing is required for correct pre-processing. Such restore can be performed by *quasi-epilogues* during non-local temporary return to the owner function as in Fig. 6.

The pseudo code for Fig. 6 is shown in Fig. 7. A call to L-closure `scan1` (*L-call* to `scan1` at `g1` in Fig. 7) starts a non-local temporary return to the owner function (`Lc0`, `Lc1`); firstly, it temporarily returns to the *selector* for the previous frame (e.g., `g3` for `gc`). Each *selector* (e.g., `o3`) selects a pre-processing branch (e.g., `o4`, `o6`) if the *current* frame is the owner of the L-closure to be called; otherwise, a quasi-epilogue branch (e.g., `o18`) is selected. Note that the quasi-epilogue branch is always taken for the functions without L-closures (e.g., `g3`, `g4` and `g6`) to continue the non-local temporary return after restoring callee-save registers (e.g., `g7`, `g8`).

The pre-processing (o6,o7,o8,o9) can be performed using the *current* frame with restored callee-save registers. After pre-processing, the control is actually transfered to the L-closure (o10, Lc2, Lc3).

Each *temporary return* finds a selector for the previous frame based on the return address for the frame; after enabling post-processing by modifying return address, it finds a selector which continues the L-call without performing further pre-processing (o16, o17).

The solid arrows for L-call to scan1 in Fig. 6 corresponds to the following steps in Fig. 7: g1, L-call (Lc0, Lc1), g1, selector (g3, g4), quasi-epilogue (g6, g7, g8), ..., o1, selector (o3, o4), pre-processing (o6, o7, o8, o9, o10), L-call(Lc2, Lc3), and scan1.

Fig. 6 also illustrates that the enabled post-processing intercepts the ordinary return. The solid arrows for the return to bin2list corresponds to the steps: getmem, post-processing (o11, o12, o13, o14), and o1.

This implementation policy for L-closures effectively decouples L-closures from their owner function, and makes the owner function's variable access faster.

5 Implementation Based on GCC

This section presents our implementation based on the GNU C compiler[6]. We enhanced GCC-3.2 to implement closures and L-closures for IA-32 and SPARC.

GCC uses an intermediate representation in Register Transfer Language (RTL) to represent the code being generated, in a form closer to assembly language than to C. An RTL representation is generated from the abstract syntax tree, transformed by various passes (such as data flow analysis, optimization and register allocation) and then converted into assembly code.

GCC has its own nested functions as an extention to C. They keep interoperability with ordinary top-level functions by using a technique called "trampolines"[7]. Trampolines are code fragments generated on the stack at runtime to indirectly enter the nested function with a necessary environment. Therefore, GCC's approach involves more creation costs than *closures*.

Closures are implemented as was mentioned in Sect. 4. We use the stack-allocated pointer pairs instead of GCC's trampolines. A pair is initialized to hold the address of the actual nested function and the static chain. To call a closure, the caller first loads the static chain (the second element of the pointer pair) into the static chain register[4] and calls the actual nested function (the first element of the pointer pair). All of these are implemented by extending only the RTL generation scheme.

On the other hand, L-closures are implemented by (1) extending the RTL generation scheme, (2) extending the assembly code generation scheme, and (3) adding short runtime assembly code. In the implementation of L-closures, we accept loss of implementation simplicity to obtain reusability (portability) and efficiency. For reusability, our implementation employs the existing RTL without modification or extension. With this approach, most existing optimizers

[4] In GCC, static_chain_rtx holds the RTL expression of the static chain register.

Table 1. Implementation costs as patches to GCC

Closures	L-Closures (+ Closures)				(lines)
RTL	RTL	IA-32(`i386`)	`i386` asm	SPARC(`sparc`)	`sparc` asm
320 lines	973	212	105	181	148

do not need modification or extension. We also minimize the extension on the assembly code generation scheme; we rather extend the RTL generation scheme if possible.

Table 1 summarizes the number of patch lines to implement *closures* and/or *L-closures* as patches to GCC. For example, the implementation of L-closure on IA-32 requires 973 line patch for RTL generation, 212 line patch for supporting selectors and quasi-epilogues, and 105 line assembly runtime code. The RTL generation part is shared with SPARC.

Since details of our implementation heavily depend on GCC internals, we only outline our implementation. We simply generate selector code and quasi-epilogue code at assembly-level by modifying the existing epilogue generation routines. Note that we assume the use of a register window for SPARC at this implementation. The pre/post-processing code is first generated as RTL code and transformed by usual optimization and register allocation phases. For the correct optimization in RTL, we employ a "virtual" control-flow edge for control transfer performed by assembly-level code such as between selector code and pre-processing code or between post-processing code and the original return point.

Our real implementation combines all selectors for each function into a single selector with all possible branches. It also employs intra-function code sharing among pre/post-processing code fragments for different call points and exploits the runtime code fragments for common tasks in pre/post-processing and quasi-epilogues. These improvements on code size produce complex code.

We do not have serious errors to use the unchanged GNU debugger to debug the generated code with L-closures; for example, the back-tracing works well. However, some execution status cannot be obtained correctly.

6 Performance Measurements

Without having nested functions, the speed of C programs will not change with our extended compiler. To measure costs of creating and maintaining lexical closures, we employed the following programs with nested functions for several high-level services and compared them with the corresponding plain C programs:

BinTree (copying GC) creates a binary search tree with 200,000 nodes, with a copying-collected heap.

Bin2List (copying GC) converts a binary tree with 500,000 nodes into a linear list, with a copying-collected heap.

fib(36) (check-pointing) calculates the 36th Fibonacci number recursively, with a capability of capturing stack state for check-pointing (see Fig. 8).

```
int cpfib(lightweight void (*save0)(), int n)
{
  int pc = 0;          /* pseudo program counter */
  int s = 0;
  lightweight void save1(){ /* nested function */
    save0();           /* saving caller's state */
    save_pc(pc);       /* saving pc state */
    save_int(n);       /* saving variable state */
    save_int(s);       /* saving variable state */
  }
  if (n <= 2) return 1;
  pc = 1;   /* inc program counter before call */
  s += cpfib(save1, n-1);
  pc = 2;   /* inc program counter before call */
  s += cpfib(save1, n-2);
  return s;
}
```

Fig. 8. Capturing state with L-closures

fib(36) (load balancing) calculates the 36th Fibonacci number, on a load-balancing framework based on lazy partitioning of sequential programs[8].

Pentomino/nqueens(13) (load balancing) perform backtrack search for all possible solutions to the Pentomino puzzle/the N-queens problem (N=13), on the load-balancing framework.

Note that nested functions are never invoked in these measurements, that is, garbage collection, check-pointing and task creation do not occur.

We measure the performance on 1.05GHz UltraSPARC-III and 3GHz Pentium 4 using `-O2` optimizers. Table 2 summarizes the results of performance measurements, where "no closures" means the plain C program without the high-level services (i.e., using no closures nor additional closure parameters for every function call). "Trampolines" means the use of GCC's conventional nested functions. In some programs, especially those creating nested functions frequently,

Table 2. Performance Measurements

S:SPARC P:Pentium		Elapsed time in seconds (relative time to "no closures")							
		no closures		Trampoline		Closure		L-closure	
BinTree copying GC	S	0.180	(1.00)	0.240	(1.33)	0.226	(1.26)	0.190	(1.06)
	P	0.150	(1.00)	0.165	(1.10)	0.167	(1.11)	0.150	(1.00)
Bin2List copying GC	S	0.289	(1.00)	0.322	(1.14)	0.292	(1.01)	0.290	(1.00)
	P	0.139	(1.00)	0.141	(1.01)	0.139	(1.00)	0.139	(1.00)
fib(36) check pointing	S	0.56	(1.00)	2.76	(4.93)	0.81	(1.45)	0.60	(1.07)
	P	0.170	(1.00)	0.468	(2.75)	0.260	(1.52)	0.170	(1.00)
fib(36) load balancing	S	0.57	(1.00)	2.46	(4.31)	0.91	(1.60)	0.68	(1.19)
	P	0.168	(1.00)	0.400	(2.38)	0.346	(2.06)	0.283	(1.68)
Pentomino load balancing	S	3.16	(1.00)	5.75	(1.82)	4.66	(1.47)	3.44	(1.09)
	P	1.80	(1.00)	2.10	(1.17)	2.06	(1.14)	1.92	(1.07)
nqueens(13) load balancing	S	0.470	(1.00)	1.022	(2.17)	0.806	(1.71)	0.592	(1.26)
	P	0.316	(1.00)	0.426	(1.35)	0.423	(1.34)	0.464	(1.47)

the speed of the conventional nested functions is less than half. In contrast, L-closures exhibits good performance. The relative times to the plain C are considerably closer to 1.00.

However, there are two exceptional results in Table 2: fib(36) and N-queens (load balancing) on Pentium 4. In these results, unimportant variables are allocated to registers. Since Pentium 4 has only a few callee-save registers and performs explicit save/restore of callee-save registers, the penalty of wrong allocation is serious. Our technique using private locations increases the number of allocation candidates, and increases not only good allocation opportunities but also wrong allocation opportunities. On the other hand, our technique is quite effective on SPARC which has more callee-save registers and performs lazy save/restore with the register window.

7 Discussion

7.1 Costs of L-Closures

Like translation techniques based on "structure and pointer"[2, 3], closures and L-closures need more code space for additional infrequently-invoked procedures than annotation-based implementations. Grouping infrequently-used procedures (plus code fragments in the case of L-closures) into a different code segment will improve locality for the instruction cache.

To scan the execution stack with n frames by the program in Fig. 2, additional n frames are needed for nested invocation of L-closures. If this is a problem, standard techniques for eliminating tail calls can solve the problem. For its time complexity, the number of *temporary returns* is $O(n^2)$. If this is a problem, we should employ another L-closure policy which always converts unconverted part of the entire stack into the pre-processed stack each time an L-closure is invoked, where only the first conversion involves $O(n)$ temporary returns in this case.

The results of performance measurements does not indicated that the cost of additional closure parameters is serious. If we can find L-closures by using *tags* like *exception handlers*, this additional cost can be eliminated.

7.2 High-Level Services: Related Work

There are at least four schemes for implementing high-level services on top of C compilers: (1) Using direct stack manipulation in C neglecting legitimacy and portability[1], (2) Providing special service routines and using the routines for the translators into C[9], (3) Using elaborate translation techniques in the translators into C[2, 3, 10, 11, 12, 13], or (4) Extending C compilers and using the extended features for the translators into the extended C[14, 15]. Our approach employs the fourth implementation scheme.

Capturing/Restoring Stack State. By using nested functions, stack state can be captured without returning to the callers.[5] Figure 8 shows a C function

[5] Restoring a previously-captured state is much easier and does not need nested functions. For restoring, different versions of C functions can be used for efficiency.

with a nested function for capturing the stack state. The program uses a pseudo program counter to record the current program point and saves all parameters/local variables. This technique can be applied to check pointing, migration and first-class continuations.

Porch[10] is a translator that transforms C programs into C programs supporting portable checkpoints. They introduce source-to-source compilation techniques for generating code to save and recover from such portable checkpoints automatically. To save the stack state, the program repeatedly returns and legitimately saves the parameters/local variables until the bottom of the stack is reached. During restoring, this process is reversed.

Multi-threads: Latency Hiding. We can implement high-level language threads realized by a language system by using L-closures. To implement multiple threads, every function has its own nested function to continue its equivalent computation and save the pointer to the nested function to be called later to early execute the thread's unprocessed computation (continuation). The explicit continuation is provided by the nested function ane explicitly passed like a *continuation-passing style*.

Concert[11], OPA[12] use similar translation techniques to support suspension and resumption of multiple threads on a single processor with a single execution stack (e.g., for latency hiding). They create a new child thread as an ordinary function call and if the child thread completes its execution without being blocked, the child thread simply returns the control to the parent thread. But in case of the suspension of the child thread, the C functions for the child thread legitimately saves its (live) parameters/local variables into heap-allocated frames and simply returns the control to the parent thread. When a suspended thread become runnable, it may legitimately restore necessary values from the heap-allocated frames.

StackThreads/MP[14] allows the frame pointer to walk the execution stack independently of the stack pointer. When the child thread is blocked, it can transfer the control to an arbitrary ancestor thread without copying the stack frames to heap. StackThreads/MP employs the unmodified GNU C compiler and implements non-standard control flows by a combination of an assembly language postprocessor and runtime libraries.

Load Balancing. 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)[16], are effective. 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.

Compilers (translators) for multithreaded languages generate low-level code. In the original LTC[16], assembly code is generated to directly manipulate the

execution stack. Both translators for Cilk[13] and OPA[12] 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*.

A message passing implementation[17] of LTC employs a polling method where the *victim* detects a task request sent from the *thief* and returns a new task created by splitting the present running task. This techniques enables OPA[12], StackThreads/MP[14] and Lazy Threads[15] to support load balancing.

We can generate an LTC-based load balancing program where callers' variables are accessed by using L-closures[8]. We can also perform *backtracking* by using L-closures. 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.

8 Conclusions

This paper has proposed a new language concept called "L-closures" for legitimate execution stack access. L-closures can be used to implement a wide variety of high-level services and can be implemented efficiently in terms of creation/maintenance costs by accepting higher invocation costs. We implemented L-closures based on GCC while reusing the existing optimizers.

The results of performance measurements exhibit quite low costs of creating and maintaining L-closures. Because most high-level services create L-closures very frequently but call them infrequently (e.g., to scan roots in garbage collection), the total overhead can be reduced significantly.

L-closures have roles similar to exception handlers, but they are sometimes more useful since they allow the control to return to the calling point.

Future work includes the implementation of various high-level languages by using our extended C language with L-closures as an intermediate language. We are also developing a transformation-based implementation of L-closures, which will be useful for the system where GCC-based compilers cannot be used.

Acknowledgments. We would like to thank Dr. Tatsurou Sekiguchi for his comments on improving the paper. This work was supported in part by MEXT Grant-in-Aid for Exploratory Research (17650008).

References

1. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Software Practice & Experience **18**(9) (1988) 807–820
2. Hanson, D.R., Raghavachari, M.: A machine-independent debugger. Software – Practice & Experience **26**(11) (1996) 1277–1299

3. Henderson, F.: Accurate garbage collection in an uncooperative environment. In: Proc. of the 3rd International Symposium on Memory Management. (2002) 150–156
4. 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)
5. Ramsey, N., Jones, S.P.: A single intermediate language that supports multiple implementations of exceptions. In: Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. (2000) 285–298
6. Stallman, R.M.: Using the GNU Compiler Collection. Free Software Foundation, Inc. for gcc-3.2 edn. (2002)
7. Breuel, T.M.: Lexical closures for C++. In: Usenix Proceedings, C++ Conference. (1988)
8. Yasugi, M., Komiya, T., Yuasa, T.: An efficient load-balancing framework based on lazy partitioning of sequential programs. In: Proceedings of Workshop on New Approaches to Software Construction. (2004) 65–84
9. Taura, K., Yonezawa, A.: Fine-grain multithreading with minimal compiler support – a cost effective approach to implementing efficient multithreading languages. In: Proc. of Conference on Programming Language Design and Implementation. (1997) 320–333
10. Strumpen, V.: Compiler technology for portable checkpoints. `http://theory.lcs.mit.edu/~porch/` (1998)
11. Plevyak, J., Karamcheti, V., Zhang, X., Chien, A.A.: A hybrid execution model for fine-grained languages on distributed memory multicomputers. In: Supercomputing'95. (1995)
12. 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) 174–188
13. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. ACM SIGPLAN Notices (PLDI'98) **33**(5) (1998) 212–223
14. 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
15. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy Threads: Implementing a fast parallel call. Journal of Parallel and Distributed Computing **3**(1) (1996) 5–20
16. 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**(3) (1991) 264–280
17. 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