Optimizing KLIC with Static Analysis

Hiroshi Nakashima  Kazuhiko Ohno
Toyohashi University of Technology
{nakashima|ohn}@tutics.tut.ac.jp

1 Introduction

Symbolic parallel languages, including KL1 and other parallel logic languages, are widely expected to solve various problems on parallel programming of non-numeric highly intelligent applications. With these languages, programmers will be comfortably free from complicated communication and synchronization among parallel objects, while PVM/MP1 users struggle with these troublesome parallel stuffs.

Along with the expectation, however, distrust in their efficiency also widely spreads and keeps people from leaving the chaotic PVM/MP1 world. Unfortunately, this “good-but-slow” notoriety is not quite unreasonable as their sequential relatives were called so a couple of decades ago. In fact an implementation of such a language may have serious performance bottlenecks caused by good features of the language which lightens programmer’s burdens.

For example, a programmer may not fork processes/threads explicitly but the run-time system has to create and schedule a large number of fine-grained parallel/concurrent objects. A complicated data structure is easily transferred between processors without packing its elements into a byte string to pass MPI_send, because the run-time system does the job paying a significant inter-processor communication cost. These heavy loads passed from the programmer to the run-time system result in inefficiency and prevent the language to be widely used in high-performance parallel symbolic computation.

Similar problems were observed in the implementation of sequential symbolic languages such as Prolog, but have been gradually solved by moving the loads from run-time to compile-time. Two decades ago, Warren showed that a Prolog program would be reasonably efficient by compiling it[1], and then proved that a careful local analysis of the program significantly improves the performance when he introduced the WAM[2]. After that, many researches have been pursued for further improvement using global static analysis to determine the mode of a variable, its type, and even the number of reference pointers to reach the variable[3]. As the result, the performance of the fastest Prolog system is now comparable to C even for the simplest list manipulation.

Thus, we observed that the global static analysis is promising to improve the efficiency of symbolic parallel languages, particularly KL1. As shown in the rest of this paper, we adopted this technique and devised two optimization method named goal threading and message packing in order to improve the performance of both sequential and parallel execution of the portable KL1 system KLIC[4]. The former is to find a set of goals to make a thread that can be executed with a fully static and sequential scheduling to reduce the overhead caused by concurrency handling. The latter is to transfer a structured data between processors gathering its elements that are really referred by the receiving processor.

In the following sections, we outline our static type analyzer and two optimization methods. Detailed discussions of them will be found in [5] and [6].

2 Type Analysis

Our two optimization methods both use the result of type analysis to determine the set of types $T(e_i)$ for each variable $e_i$ in a program. Each element $e_j$ of $T(e_i)$ is a tuple that has a member $t_j$ representing that $e_i$ can be instantiated to a term whose type is $t_j$. The type $t_j$ is one of the following.
/*1*/ sorter(S):- S = [ ] | true.
/*2*/ sorter(S):- S = [X|Sn],
    M = do(L,R),
    RT = [ ],
    qsort(L,R,RT),
    sorter(Sn).
/*3*/ qsort(L,RH,RT):- L = [ ] | RH = RT.
/*4*/ qsort(L,RH,RT):- L = [X|Ln] | partition(Ln,X,LL,LM),
    RH = [X|RHM],
    qsort(LL,RH,LM),
    qsort(LG,RHM,RT).
/*5*/ partition(L,X,LL,LM):- L = [ ] | LL = [], LG = [].
/*6*/ partition(L,X,LL,LM):-
    L = [Y|Ln], Y <= X | LL = [Y|LLn],
    partition(Ln,X,LLn,LM).
/*7*/ partition(L,X,LL,LM):- L = [Y|Ln], Y >= X | LG = [Y|LGN],
    partition(Ln,X,LL,LGN).

Figure 1: Sample Program sorter/1

- a for an atomic term.
- s(f,n,(T₁,...,Tₙ)) for a compound term f/n whose k^{th} argument will have one of the types in the set Tₖ.
- n for a variable occurring in a compound term.
- * for a term whose type is unknown at a phase, which may be the final one, of the analysis.

For example, we will have the following analysis results from the program fraction shown in Figure 1.

\[
\begin{align*}
T(S₁) &= T(RT₂) = T(L₃) = T(L₅) = T(LL₅) \\
      &= T(LG₅) = \{a\} \\
T(X₄) &= T(X₁) = T(X₅) = T(Y₅) \\
      &= T(Y₇) = \{a\} \\
T(S₇) &= \{s(\cdot,\cdot,2,(\{X₂\},\{Sn₂\}))\} \\
T(Sn₉) &= \{a, s(\cdot,\cdot,2,(\{X₂\},\{Sn₂\}))\} \\
T(R₂) &= \{do,2,(\{L₂\},\{R₂\})\} \\
T(R₄) &= T(RH₃) = T(RT₄) = T(RH₄) = T(RT₄) = T(RMN₄) \\
      &= \{a, s(\cdot,\cdot,2,(\{X₄\},\{RHN₄\}))\} \\
T(RMN₄) &= \{s(\cdot,\cdot,2,(\{X₄\},\{RHN₄\}))\} \\
T(L₂) &= T(LL₄) = T(LL₅) = T(LL₅) = T(LL₅) \\
      &= T(LG₇) = T(LG₇)
\end{align*}
\]

Note that the program is canonicalized from its original form like:

\[
\text{sorter([do(L,R)|Sn]):}- \text{true} \mid \ldots
\]

and the variable Yₖ is Y in the k^{th} clause.

To obtain the results, our analyzer at first finds out local type constraints and then propagates them from callers to callees and vice versa through their arguments. For example, T(L₄) initially has the set containing only one type \(s(\cdot,\cdot,2,(\{X₄\},\{L₄\}))\) for the cons \([X|Ln]\). Then the set is propagated to the callers of the fourth clause, which are two goals qsort/3 in itself and that in the second clause, through the first argument of qsort/3. In the propagation from a callee to callers, the set is unionized with the sets for arguments of other callees to represent that a caller’s argument may have other types. Thus the callers argument, e.g. L₂, will have another type a for [] propagated from the third clause. This unionization is also performed in the caller-tocallee propagation.

Variables occurring in a compound term are preserved in the propagation and unionization in order to keep a type representation as complex as its corresponding term directly described in the program. This also ensures that the analysis will terminate. Thus the cons type in the final result for L₂ has three variables each in car and cdr parts although they eventually have same types, an atomic (integer) for car and a cons or nil for cdr recursively.

This analysis with constraint-satisfaction is originated in the mode analyser of Ueda and Morita for flat-GHC/K.L1'[7], and has the following advantages over abstract interpretation adopted by most of type/mode analyzers for logic languages[8]. The first
client(..., S):-
    "terminate condition" | S = [].
otherwise.
client(..., S):- true |
    make_list(..., L),
    S = [do(L,R)|Sn],
    use_result(r, ...),
    client(..., Sn).

Figure 2: client of sorter/1

one is its time-independent nature well suitable to non-deterministic concurrent execution of KL1 programs. The second and more important advantage is its adaptability to a fraction of a program. This not only gives us the basis of separate compilation but also is essential for our message packing as described later.

3 Goal Threading

3.1 Problems in Conventional Goal Scheduling

Since KL1 has AND-parallel semantics, any order of goal reduction will give a correct result\(^\dagger\). Thus the sequential core of a KL1 run-time system may have its own strategy of the goal scheduling expecting the strategy fits to most of programs. A straightforward strategy is called process-oriented scheduling adopted for Multi-PSI and PIM\(^9\), which reduces goals in depth- and left-first order until a goal suspends because its arguments are not sufficiently instantiated to satisfy the constraints given in the guards of its matching clauses\(^2\). This strategy is quite simple and almost optimal for sequential programs. For example, qsort/3 shown in Figure 1 will be executed without any scheduling overhead providing its first argument is fully instantiated.

The drawback of this simple strategy,

\(^\dagger\)This does not mean the result of a program is independent of the reduction order.

\(^2\)KL1 has a scheme to give a priority to a goal and thus the run-time scheduler takes care of it by preempting a highly prioritized goal when the goal becomes ready. This mechanism, however, is not essential in this discussion and thus is ignored in this paper.

however, is revealed when a program has concurrency. Suppose the program fragment shown in Figure 2 and a clause containing

    ..., client(..., S), sorter(S), ...

to connect client with sorter/1 shown in Figure 1 are given. This programming style, in which processes are recursive predicates and streams for inter-process communication are incomplete lists of also incomplete structures, is frequently used not only to describe concurrent processes but also to give object-oriented flavor to KL1 programs.

The process-oriented strategy shows its inefficiency in the execution of this client-sorter program. The run-time system tries to reduce client completely but will find that use_result suspends waiting R that ready but unscheduled sorter will instantiate. Then it further tries to reduce the recursive goal of client blindly, simply to find that the goal immediately suspends if its termination condition depends on the output from use_result as in usual programs. At this time, the system at last picks sorter and executes qsort smoothly, but tries and fails to reduce the recursive goal of sorter again. Finally, the control returns to use_result that is now reducible and this scenario is repeated. Note that the suspension occurs in both side in one cycle, twice in client and once in sorter.

In order to reduce the number of suspensions, KLIC employs a slightly modified strategy named resumption-first in which a goal suspended with variable v is preemptively scheduled when v is instantiated. This strategy makes the suspension in the client-sorter single-sided. The execution scenario of the resumption-first strategy is same as that of the process-oriented in the first half, and use_result and client suspend. In the second half, however, qsort in sorter instantiates R of use_result that is then immediately resumed. Thus the further reduction of sorter and its suspension are avoided because the control will not return to sorter’s side until client suspends again.

Although it would seem clear that the
resumption-first is better than the process-oriented because of the number of suspensions, a careful examination of the scenario shows the resumption-first may behave much more poorly. Remember an instantiation of a variable does not mean that it turns to having a ground term, rather it may have a incomplete structure containing uninstantiated variables. In fact, when the terminating clause of qsort instantiates R and resumes use_result, R has a cons of the smallest element in car and an uninstantiated variable in cdr which will be instantiated by the following reduction of qsort. Therefore, use_result will immediately suspend in its second iteration, another qsort is scheduled to supply the second element and to resume use_result again, and this terrible scenario repeats for each element of the sorted list\(^3\).

Similar ping-pong scheduling will be observed when both client and sorter suspend waiting a variable to be instantiated by another process. In this case, when client is resumed it at first performs S=[do(L,R)|Sn] before trying to reduce make_list and other goals. Thus sorter is resumed by the unification of S before make_list supplies the value of L, and its goals qsort and sorter immediately suspend. After that, make_list is scheduled and supplies the first element of L causing the resumption of qsort which simply results immediate suspensions of partition and two recursive qsort, and this ping-pong is repeated for each element of L.

### 3.2 Improvement by Threading

We observed it is the reason of the inefficiency of conventional scheduling strategies that they blindly believe the concurrency of goals. For example, even if partition suspends on its first iteration, these strategies try to reduce two recursive qsorts expecting they concurrently work with partition. Against this expectation, however, these goals of course immediately suspend waiting results from partition. If partition suspends on its \(n^{th}\) iteration, qsorts can do some works but finally create approximately \(3n\) suspending goals. Since these suspensions and corresponding resumptions in future are much more costly than usual reductions, choosing a goal other than qsorts is obviously better scheduling.

Another important fact found in this example is that qsorts are not necessary to be scheduled until partition completes. This means that qsorts cannot contribute to the resumption of suspended partition. Thus we now conclude that the goals in the recursive clause of qsort/3 can be scheduled in a sequential depth- and left-first manner, in which a goal is scheduled after its left-sibling goal completes. We call such goals being in a thread, which is the unit of the concurrent execution in place of goals.

To make a thread from goals, we have to analyze the dependency of the goals derived from the data flow among them. The data flow is obtained by a type analyzer, which is based on that shown in Section 2 but is modified in the following points.

1. The initial local constraints are obtained only from the output arguments of goals of built-in predicates in body parts, including \(/\)/ having a non-variable argument.

2. Each element \(e_j\) of the set of types \(T(v_i)\) for variable \(v_i\) in clause \(c\) is a tuple \((t_j,G_j)\) of the type \(t_j\) shown in Section 2 and a set of goals \(G_j\) whose element \(g\) is one of the following\(^4\):
   
   (a) \(g = 0\) if \(t_j\) is derived by the caller-to-callee propagation through the clause head of \(c\).
   
   (b) \(g = k > 0\) if \(t_j\) is derived from the \(k^{th}\) body goal in \(c\) as the initial constraint or by the callee-to-caller propagation.

Additionally, if \(t_j\) is a variable \(v_i\) occurring in a compound term, \(v_i\) is denoted as \(V_{c/m}^d\) representing that \(V\) occurs in

---

\(^3\)This disaster is avoidable by exchanging two recursive goal of qsort as in the program included in KLIC's distribution.

\(^4\)The real implementation shown in [5] is slightly different from that shown here for efficiency.
client(N,S) :- N = 0 | S = [].
client(N,S) :- N > 0 |
    S = [M|Sn], M = do(L,R),
    integers(N,L),
    I = 0,
    check(R,I,N,Hn),
    client(Hn,Sn).

integers(N,L) :- N = 0 | L = [].
integers(N,L) :- N > 0 |
    L = [N|Ln], Hn := N - 1,
    integers(Hn,Ln).

check(R,I,N,Hn):- R = [] | Hn := N - 1.
check(R,I,N,Hn):-
    R = [J|Hn], I =: J |
    In := I + 1,
    check(Hn,In,N,Hn).

main:-
    N = 100, client(N,S),
    sorter(S).

Figure 3: Simple client and main

the clause d but is the mth local version
for the clause c.

For example, the analysis results for the variables in the first recursive clause of
\text{partition/4} (clause #6 in Figure 1) are the following with a simple client and main
shown in Figure 3.

\begin{align*}
T(L^4) &= \{(a',\cdot,2,\langle x_{i_j}^1,\{L_{i_j}^1\}\rangle,\{0\})\} \\
T(L_{i_j}^1) &= T(x^4) = T(y^4) = \{(a,\{0\})\} \\
T(Ln^4) &= T(L_{i_j}^1) = \\
&= \{(a,\{0\}), \\
&\langle a',\cdot,2,\langle x_{i_j}^1,\{L_{i_j}^1\}\rangle,\{0\}\rangle\} \\
T(LL^4) &= \{(a',\cdot,2,\langle x_{i_j}^1, \\
&\{LLn^4,\{L_{i_j}^1\}\}\rangle,\{1\}\}\} \\
T(LLn^4) &= T(LL^4) = T(LLn_{i_j}^1) \\
&= \{(a,\{2\}), \\
&\langle a',\cdot,2,\langle x_{i_j}^1, \\
&\{LLn^4,\{L_{i_j}^1\}\}\rangle,\{2\}\rangle\} \\
T(H_{i_j}^1) &= \{(a,\{0,2\})\}
\end{align*}

With the results of the data flow analysis, dependency among sibling goals is checked. A
goal g is potentially dependent to its sibling goal g' if T(A) for an argument A of g
has the type tuple \((t_{i_j},G_j)\) such that \(g' \in G_j\). If \(t_{i_j}\) is a compound term, this condition is
recursively applied to its arguments. Then the goal g is really dependent to g' if a guard
goal of g or its descendents requires that A

has the type \(t_{i_j}\), and we denote this dependency as \(g' \prec g\). For example, the goals in
the recursive clause of \text{qsort/3} (clause #4 in Figure 1) have the following dependency.

\begin{align*}
\text{partition}(Ln, X, LL, LG) &\prec \text{qsort}(LL, RH, RH) \\
\text{partition}(Ln, X, LL, LG) &\prec \text{qsort}(LG, RMn, RT)
\end{align*}

Note that among two \text{qsorts} and \(=\)/2 there are potential dependency but no real dependency
because their guard goals don't refer data generated by them.

The dependency gives us a necessary condition for threading. A clause is locally
threadable if it has no pair of body goals g and g' such that \(g' \prec g\) and \(g' \prec g\). Thus all the clauses in Figure 1 and client/1, integers/2 and check/2 in Figure 3 are
locally threadable. The clause \text{main/0}, however, is not locally threadable because

\text{client}(S) \prec \text{sorter}(S) and \text{sorter}(S) \prec \text{client}(S).

This unthreadable clause introduces the necessity of further analysis of the threadability,
because it may reveal a hidden dependency among goals in a locally threadable clause. For example, the goals \text{qsort}
and \text{sorter} in the clause of \text{sorter} are apparently independent but has the dependency of \text{qsort}(L,R,RT) \prec \text{sorter}(Sn).

This global threadability analysis is hardly precise because it is required to unfold a
recursively structured term so that each instance of the term is handled individually.
Since complete unfolding is obviously impossible, we limit the number of unfolding
to a small constant, in fact one, with which most of programs including this example are properly threaded.

As the result of the analysis, the goals in a threaded clause is reordered to satisfy the
dependency among the goals if necessary. As for a unthreadable clause, a pragma
\text{@thread} is added to one of the mutually dependent goal pair in order to notify the runtime system to create a new thread. In this example, the goal reordering is unnecessary
but the clause of \text{main} is modified as follows.

\begin{verbatim}
main:-
    N = 100, client(N,S),
    sorter(S)@thread.
\end{verbatim}
Since a thread is executable in depth and left-first order, its execution environment may be managed with a stack, rather than the linked list of goal records on heap which KLIC’s run-time system manages. The management with stack will not only reduce the frequency of garbage collection but also improve the access locality for local variables. Thus it is expected that the threading will improve the performance of both concurrent and sequential portions of a program.

The performance results shown in Table 1 meet this expectation. Two to three fold speedup is achieved for two concurrent programs `handshake` and `reverse` by the reduction of suspensions. As for other sequential programs, the speedup is not drastic but still significant, except for `nrev` whose execution behavior is almost unchanged by threading.

<table>
<thead>
<tr>
<th>Program</th>
<th>Original KLIC</th>
<th>Threaded KLIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>S</td>
</tr>
<tr>
<td>handshake</td>
<td>19.21</td>
<td>1000</td>
</tr>
<tr>
<td>reverse</td>
<td>79.42</td>
<td>2964</td>
</tr>
<tr>
<td>nrev</td>
<td>0.199</td>
<td>0</td>
</tr>
<tr>
<td>kkqueen</td>
<td>0.560</td>
<td>0</td>
</tr>
<tr>
<td>factorial</td>
<td>0.214</td>
<td>0</td>
</tr>
</tbody>
</table>

T: execution time (sec)  
S: # of suspensions ($\times 10^3$)  
G: # of garbage collections  
U: speedup

4 Message Packing

4.1 Problems in Conventional Message Transfer

In the distributed implementation of KLIC, communications between goals allocated on different processing nodes are performed with a reader-oriented asynchronous message passing mechanism. For example, if the clause of `main` shown in Figure 3 is;

```
main:-
    N = 100, client(N,S),
    sorter(S)@node(1).
```

the goal `sorter` is allocated on the processing node #1 ($N_1$), while `client` stays in the default node #0 ($N_0$).

Since the variable $S$ is allocated on $N_0$, the run-time system on $N_1$ will find the argument $S$ of `sorter` has an external reference pointer to $S$ on $N_0$ when it tries to reduce the goal. Thus $N_1$ sends a `%read` message with the pointer to $N_0$ to get the value of $S$ that is required for the choice and commitment of a clause of `sorter`. The message is received by $N_0$ that examines the value of $S$ and send a `%answer_value` back to $N_1$ with the encoded value of $S$ if it is instantiated. Otherwise, the replying operation is suspended until the instantiation of $S$ by a mechanism similar to goal suspension. Similarly, $N_0$ will find the argument $R$ of `check` is external and thus messages are transferred between $N_0$ and $N_1$ exchanging their roles$^5$.

In these scenarios, it is not clear what portion of the structured value of $S$ or $R$ is replied through `%answer_value`. KLIC has two different mechanisms to make the reply, default lazy and optional eager mode chosen by a command switch[10, 11].

In the lazy mode, the replier node sends back the `surface` value of a externally referred variable if it is instantiated to a compound term. That is, if the variable has the term $f(A_1, \ldots, A_n)$ and $A_i$ is also a compound term, `%answer_value` carries the external reference pointer to $A_i$ instead of its value. If the requester node needs the value of $A_i$, it sends `%read` again to get the value. This mechanism minimizes the total amount of data value transferred between nodes because the transfer is completely on-demand.

However, if the requester needs to have the whole of a structured data, the number of messages and the latency to obtain it may be terribly large. For example, the guard of the recursive clause of `sorter` requires that $S$ is instantiated to the form of `[do(L,R)|Sn]` that is obtained by two `%read/%answer_value` sequences, first for

$^5$: The very first cons of the list is sent from $N_1$ by a `%unify` message because the $R$ in `client` is allocated on $N_0$. 

[M|Sn] and then for do(L,R). Much worse than that, the reduction of qsort refers the whole of the list of length n in L causing message exchanges n times. The scenario to get R for check is as bad as it.

The eager mode will solve this problem because an externally referred structure is completely traversed to its leaves, atomic values or uninstantiated variables, and all of its instantiated portions are encoded and carried by one \texttt{answer_value} providing the encoding result fit to the buffer of predefined size. Thus as the reply for \texttt{read} to S of \texttt{sorter}, N_1 will receive \texttt{answer_value} with;

\texttt{do([n, \ldots, 1], R)|Sn}

which is enough to one recursion of \texttt{sorter}.

This mechanism, however, introduces another problem due to the fact that it ignores whether a portion of a structure is really referred by the requester. Suppose each car of the list to be sorted is not an integer but a compound term \texttt{record(K,V)} where K is an integer sorting key and V is an arbitrary complicated and large term. Although \texttt{sorter} is not refers V because its \texttt{partition} should be like;

\texttt{partition(L,X,LL,LM):-}
\texttt{L = [record(Y,V)|Ln], Y < X |}
\texttt{LL = [record(Y,V)|LLn],}
\texttt{partition(Ln,X,LLn,LM).}

the whole of V will be sent paying unnecessary encoding overhead and wasting the bandwidth of the channel connecting nodes. Moreover, if V is too large to encode into a single message, the number of messages is proportional to the length of the list.

4.2 Improvement by Packing

Our approach to solve the problems is to make the eager mechanism clever so that only the really referred portions of a structure are sent back to the requester. To know whether a portion is referred, we use a modified version of type analysis again.

First, a program is divided into (possibly overlapped) subprograms for parallel processes whose initial goals are \texttt{main} or goals with \texttt{@node} pragma. We refer to each subprogram and parallel process as P_g where g is its initial goal. From the example shown in Section 4.1, we will obtain a subprogram P_{main} containing \texttt{main}, \texttt{client} and the descendents of \texttt{client}, and P_{sorter} of \texttt{sorter} and its descendents.

Then the type analysis is performed on each subprogram independently, together with mode analysis using an algorithm similar to that of Ueda and Morita[7]. Since only a part of the program is analyzed, the result may be insufficient to know the types of all the variables in a subprogram. This insufficiency, however, is important to detect the reference of a structure.

From the results of each analysis, only the results for the arguments of goals with \texttt{@node} pragma are picked for further works. In our example with the list of key/value pairs, T(S) for P_{sorter} will be equivalent to the following informal notation, where + and - represent input and output mode with respect to P_{sorter} and * means unknown by the analysis.

\texttt{[+do([record+(a,+),s,...],}
\texttt{[-record-(a,-),s,...],...],}

Similarly, the result for P_{main} is as follows providing V of record is a compound term f/N.

\texttt{[-do-([record+(a-,f-),s,...]),}
\texttt{[record+(a+,f+),s,...],...],}

These results show that the portions which are known as input mode are necessary to be carried by \texttt{answer_value}. It is important that the type of V of record is unknown in the result for P_{sorter}.

Finally, a KL1 code to traverse and encode referred portions is generated for each argument of each goal with \texttt{@node} in each subprogram. This \texttt{packing code}, C_{sorter} for example, is executed when a node receives \texttt{packed_read} message with the pointers to a variable and the code. While the execution, if an uninstantiated variable required to send is found, the pointers of the variable and the corresponding code in cluded in C_{sorter} are put in the reply for the communication in future. As for an output portion, the pointers are also put but the code
pointer refers an entry in $C_n$ for the future communication in reverse direction.

Table 2 shows the performance results of the message packing obtained by an evaluation on a multicomputer AP1000. The message packing is much better than the lazy mechanism and achieves two to three fold speedup due to the drastic reduction of the number of messages. In usual cases, the speedup from the eager mechanism is smaller, 8 to 42%, because the reduction of the total amount of transferred data is not huge. In the case of mastermind, however, a drastic 18-fold speedup is achieved proving that the eager mechanism cannot be applied generally.

5 Conclusion

We discussed about our two optimization methods for KLL, goal threading and message packing, both of which exploit the result of global type analysis. Performance results show that these methods are widely applicable to improve both sequential and parallel performance of KLL programs.

We are now trying to extend these methods in terms of both sequentaility and concurrency of programs. For example, the execution speed of a thread will be further improved exploiting its deterministic behavior, while its potential concurrency may be utilized for parallel performance tuning.

Acknowledgments

We would like to thank to Masahiko Ikawa and Namiyo Sugiyama who greatly contributed to the implementation of our optimization methods. We also thank to the members of Tomita Lab. in Kyoto University and KLLC working group for their helpfull comments. The research works presented in this paper are partly supported by the funds from ICOT and AITEC of JIPDEC.

References