PACT '94: International Conference on PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, August 23-26, 1994, Montreal, Canada

The Plan-Do Style Compilation Technique for Eager Data Transfer in Thread-Based Execution

M. Yasugi[a], S. Matsuoka[b] and A. Yonezawa[a]

[a]Department of Information Science, Faculty of Science, the University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113

[b]Department of Mathematical Engineering, Faculty of Engineering, the University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113

Abstract: *Plan-do* compilation technique is a new, advanced compilation framework for *eager data transfer* on distributed-memory parallel architectures. The technique is especially effective for a recent breed of low-latency architectures by realizing a high-throughput low-latency communication scheme, *pipelined sends*. The compilation of high-level, plan-do style code into low-level, eager data transfer code is achieved via straightforward application of a set of translation rules. Preliminary low-level benchmark results on a real parallel architecture, EM-4, exhibit good speedups.

Keyword Codes: C.4.2; D.1.3; D.3.4
Keywords: Multiprocessors; Concurrent Programming; Language Processors

1 Introduction

Distributed-Memory Massively Parallel Processors (DM-MPPs) have attractive features for scalability. There are many types of DM-MPPs; but, irrespective of their architectures, the basic abstract execution models of such machines would be those of *asynchronously communicating (multiple) threads.* Such an abstract machine could be naively realized by adding message-passing routines to the thread-based abstract machines; the collection of those routines is usually called a message-passing library.

However, such a library-based approach prevents us from exploiting advanced implementation techniques for communication; such techniques can be exploited only by compilation-based approaches. They include, for example, a technique of compiling message interpretations, known as *active messages*[6], in which a message omits the runtime tags for the message format and carries the address of a compiled message handler. There are further advanced implementation techniques enabled by compiling communications.

In this paper, we concentrate on an implementation technique, *eager data transfer*, and describe its compilation framework. The idea of *eager data transfer* originates from the dataflow execution model, but we employ it in the context of thread-based execution models. Eager data transfer can be (re)stated as "eagerly sending a datum to the location where the datum will be used," which improves both throughput and latency by reducing local buffering of messages on both sender and receiver nodes. Of course, there is a tradeoff for the excessive use of eager data transfer; it increases the number of messages and matchings of the messages, but we demonstrate the effectiveness when employed with compiled *pipelined sends*, via performance measurements on a fine-grained hybrid parallel architecture EM-4[4, 3].

There are many different levels of abstraction, even for the same general notion of 'directly communicating threads.' For example, Concurrent Object-Oriented Programming (COOP) languages, such as ABCL[10], provide high-level abstraction, while distributed memory machines themselves provide the lowest-level of abstraction. In this paper, we regard the compilation process as shifting down the abstraction levels within the directly communicating threads model, and discuss the necessary compiler supports.

In order to realize eager data transfer, dataflow information is indispensable. The compiler must provide additional dataflow information at every level of the abstraction; i.e., every intermediate representation must embody dataflow information to maintain modularity (clear separation of layers), even at the levels where optimization techniques based on dataflow information are not applied. This requirement causes a difficulty in combining several compilation phases into one pass. The *plan-do style* intermediate code solves this problem by representing the combination of control-flow information and dataflow information in a stream form. Furthermore, it has an additional benefit of serving as a convenient intermediate representation when pipelined parallelization of the compilation phases themselves is desired.

2 Eager Data Transfer

In order to clarify our motivation, this section presents several examples of eager data transfer, which can be used at various abstraction levels.

2.1 A Fine-Grained Example — Pipelined Sends

A (remote) message passing implementation scheme which overlaps computation and communication, *a pipelined send*, has been proposed for ABCL/EM-4[8, 9] (a COOP language system ABCL on EM-4). This scheme overlaps the evaluation of message arguments and their sending in a pipelined way; it improves both throughput and latency by eliminating local buffering of messages on both sender and receiver nodes. The scheme is realized by (1) remote code invocation, (2) remote write, and (3) FIFOness of network, which are efficiently supported by EM-4.

The packet exchange protocol for the pipelined message passing proceeds as follows (See Figure 1): (1) The sender reserves a message box on the receiver node, by invoking a proper code block to return the allocated message box address. (2) The sender evaluates the arguments of the message. As each argument is evaluated, its packet is eagerly sent to the proper location in the message box on the receiver node. The evaluation and sending are pipelined in a fine-grained manner. (3) The message box address is sent to the receiver. It may enqueue the message until it can process the message.

Although packet transmissions to reserve a message box may result in a round-trip latency, multi-threading at the sender node effectively hides the latency. Moreover, the reservation provides us the following significant advantages: (1) It allows us to manage the message queue efficiently at the receiver node, just by pointer manipulation rather than copying all the contents of the message. (2) The knowledge of the target message box address makes pipelined send possible, which reduces (a) *the communication latency* by eager data transfer before the complete evaluation of the message and (b) *the cost of*

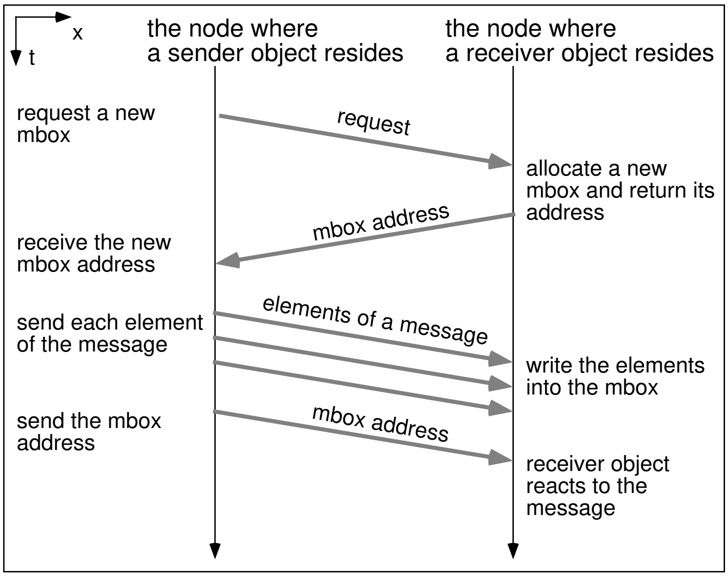

Figure 1: Packet Exchange for a Pipelined Request Message Send

local memory access for local buffering at the sender node. Moreover, pipelined sends can be interruptible, namely, multiple sends can be interleaved with one another.

2.2 Coarser-Grained Examples

Eager data transfer can also be exploited at coarser levels of granularity:

```
[Obj1 <= [:Msg1 [:cons 1 [:cons 2 [:nil]]]]]
```

This program fragment in a COOP language, ABCL, denotes a request message send to `Obj1`, whose message consists of a message tag `:Msg1` and a list of integers `1` and `2`. Suppose that this is a remote message send and *cons cells* are used to implement the list. We would like the compiler to generate the code for *remote cons* at the node where the `Obj1` resides, rather than *local cons* plus *remote copy*.

This kind of eager data transfer is not limited to COOP languages. The corresponding example in a (thread-based) functional language would be as follows:

```
(Fun1 [:cons 1 [:cons 2 [:nil]]])
```

Here, we would like the compiler to generate the code for *remote cons* at the node where the function invocation frame for `Fun1` resides.

3 Compilation Framework for Eager Data Transfer

The high-level code should contain certain amount of dataflow information to realize eager data transfer at the lower-level. The dataflow information is actually needed for code conversion (generating lower-level code from higher-level code), rather than for mere code optimization. The information includes (when fixing a value v): (1) whether or not v is used eventually, (2) which thread (node) will use v, (3) to which location on memory v will be finally stored, etc. Both (2) and (3) are especially indispensable for realizing the pipelined sends described in Section 2.1.

3.1 Inlining Dataflow Information in Plan-Do Style

The *plan-do style* intermediate code satisfies the requirement described above and also provides additional benefits for the compilation process itself. This subsection briefly describes the plan-do style code; the important effect of using the plan-do style (i.e., realizing eager data transfer) is demonstrated when the plan-do style code is interpreted by the lower-level part of the compiler, which is presented in Section 3.2.

The basic idea behind plan-do style is that (1) the *plan-part* (i.e., plan declaration part) provides dataflow information, and (2) the *do-part* (i.e., plan execution part) provides control-flow information. The benefits of the plan-do style are: (1) it takes a stream form instead of an arbitrary graph structure, and (2) interleaving of plan declaration and plan execution is possible. Thus, it allows combining of several compilation phases into one pass, or alternatively, pipelined parallelization of the compilation phases.

There are compilation targets for which the plan-do style code is specially suitable, which are the *expressions* representing natural dataflow information; using the plan-do style code makes it possible to extract and represent the information without costly flow analysis. The plan-do style code embodies the semantics of the source program in terms of sequential execution of a single thread of control, and also provides dataflow information without loss of machine independence of high-level code.

Let us take a simple message send example:

```
[obj1 <= [(+ i 1) (+ j 2)]]
```

This program fragment in ABCL denotes a request message passing to object `obj1`, whose message is a tuple of an integer `i`+1 and an integer `j`+2. This source program is converted into the following plan-do style code via the high-level parse tree:

```
(newplan (p1 d1 d2) (request-send))
(throw obj1 d1)
(newplan (p2 d3 d4) (make-tuple d2))
(newplan (p3 d5 d6) (arith3-+ d3))
(throw i d5)
(throw 1 d6)
(do p3)
(newplan (p4 d7 d8) (arith3-+ d4))
(throw j d7)
(throw 2 d8)
(do p4 p2 p1)
```

`(newplan (p1 d1 d2) (request-send))` declares *plan* `p1` together with *destinations* `d1` and `d2`. By 'destination,' we mean an argument of the plan (which corresponds to a dataflow arc to the plan node in terms of dataflow model). The plan is to send a *request message* to a target object. The target object's name is given by 'throwing' the name to `d1`. By 'throwing,' we mean that a value is passed as an argument for a plan. `(throw obj1 d1)` throws the value of `obj1` to `d1`. `(do p1)` carries out the plan `p1`. Similarly, the actual parameter of the request message is provided by throwing the obtained value (i.e., a tuple consisting of `i`+1 and `j`+2) to `d2` in a subsequent execution of another plan.

Accordingly, `(newplan (p2 d3 d4) (make-tuple d2))` declares plan `p2` to (1) make a tuple of values thrown to `d3` and `d4` then (2) throw the tuple to `d2`. `(newplan (p3 d5 d6) (arith3-+ d3))` declares plan `p3` to (1) add values thrown to `d5` and `d6` then (2) throw their sum to `d3`.

The translation of the parse tree into the plan-do style code is straightforward: When the (recursive) translation function reaches a certain tree node, before the translation of the subtrees, it declares the plan for the current tree node, and after the translation of the subtrees, it 'executes' the plan.

The standard (trivial) interpretation of the plan-do style code (which is not employed for eager data transfer) is as follows: (1) A plan declaration is interpreted just as declaration and no actions are performed. A destination is interpreted as a temporary variable. (2) A 'throw' is interpreted as an assignment of the value to the corresponding temporary variable. (3) A plan execution is interpreted as an actual execution of the plan. In Section 3.2, an alternative interpretation developed for eager data transfer is presented.

3.2 Interpreting Plan-Do Style Code to Generate Eager Data Transfer Code

This section describes the compilation process for realizing eager data transfer, by interpreting the high-level plan-do style code. We exemplify the process for pipelined sends described in Section 2.1 with the simple message send example in the previous section.

Figure 2 shows the translation function *Tr* from the high-level code into the lower-level code. Here, when *hcode* is high-level plan-do style code, $Tr\,[\![hcode]\!]\,(penv, denv)$ returns the lower-level code in the context of the plan environment *penv* and the destination environment *denv*. *penv* is a function from a set of plan identifiers (ranged over by p) to plan definitions (p corresponds to a dataflow node in terms of the dataflow model). *denv* is a function from a set of destination identifiers (ranged over by d) to pairs of p and integers (ranged over by i). (d stands for i-th dataflow arc to p in terms of the dataflow model.) $Trnsfr\,[\![v_l, fl, d]\!]\,(penv, denv)$ returns a fragment of lower-level code which transfers a value v_l to the *fl*-field of destination d. The results of *Trnsfr* are significantly influenced by the dataflow context of (*penv*,*denv*) in order to select the eager data transfer instructions under certain conditions. For a function f, "$f\{x \mapsto y\}$" denotes the function f' such that $Dom(f') = Dom(f)$, $f'(x) = y$ and $f'(x') = f(x')$ for all $x' \in Dom(f) - \{x\}$. By "$h :: r$," "::" denotes consing of the head h and list r, and "$[a, b, c]$" denotes the list consisting of a, b, and c. By "l_1 @ l_2," "@" denotes the concatenation of two lists l_1 and l_2.

The translation function basically translates the high-level code in the following way:

- For a plan declaration, it memorizes the plan and the destinations in environments *penv* and *denv*. Fresh lower-level variable identifiers will be introduced if necessary.
- For a plan execution, it generates code for finishing the plan, then if the result value has not been eagerly sent, it generates data transfer instructions by calling the *Trnsfr* function.
- For throwing a value to a destination, it decomposes the (possibly) nested higher-level tuple value into a set of pairs of lower-level values and the field number lists by calling *Decomp*, and generates data transfer instructions by calling *Trnsfr*.

Tr and *Trnsfr* embody the translation into eager data transfer; i.e., they realize the pipelined sends. For this purpose, *Tr* translates the high-level code in the following way:

- For request message passings, it prepares pointers for a request message box to realize a pipelined send, and it only generates instructions for pointer manipulation.
- For tuple construction, it eagerly transfers the tuple elements without buffering.
- For simple arithmetic operations, it simply prepares three temporary variables for the operation, and the result of the operation is transferred by *Trnsfr*.

$$
\begin{array}{l}
Tr[\![\texttt{(newplan } p_1 \; d_1 \; d_2 \texttt{ (req-send))} :: r]\!]\,(penv, denv) = \\
\qquad Tr[\![r]\!]\,(penv\{p_1 \mapsto \textbf{req-send}(t_1,t_2,t_3)\}, denv\{d_1 \mapsto (p_1,1), d_2 \mapsto (p_1,2)\}) \\
\qquad (t_1,\, t_2,\, t_3 \text{ new}) \\
Tr[\![\texttt{(newplan } p_1 \; d_1 \; d_2 \texttt{ (make-tuple } d_3\texttt{))} :: r]\!]\,(penv, denv) = \\
\qquad Tr[\![r]\!]\,(penv\{p_1 \mapsto \textbf{make-tuple}(d_3)\}, denv\{d_1 \mapsto (p_1,1), d_2 \mapsto (p_1,2)\}) \\
Tr[\![\texttt{(newplan } p_1 \; d_1 \; d_2 \texttt{ (arith3-+ } d_3\texttt{))} :: r]\!]\,(penv, denv) = \\
\qquad Tr[\![r]\!]\,(penv\{p_1 \mapsto \textbf{arith3-+}(d_3,(t_1,t_2,t_3))\}, denv\{d_1 \mapsto (p_1,1), d_2 \mapsto (p_1,2)\}) \\
\qquad (t_1,\, t_2,\, t_3 \text{ new}) \\
Tr[\![\texttt{(do } p_1\texttt{)} :: r]\!]\,(penv, denv) = \\
\qquad \texttt{(get-read-pointer } t_2 \; t_3\texttt{)} :: \texttt{(req-send } t_3 \; t_1\texttt{)} :: Tr[\![r]\!]\,(penv, denv) \\
\qquad \text{if } penv(p_1) = \textbf{req-send}(t_1,t_2,t_3) \\
Tr[\![\texttt{(do } p_1\texttt{)} :: r]\!]\,(penv, denv) = Tr[\![r]\!]\,(penv, denv) \qquad \text{if } penv(p_1) = \textbf{make-tuple}(d_1) \\
Tr[\![\texttt{(do } p_1\texttt{)} :: r]\!]\,(penv, denv) = \\
\qquad \texttt{(arith3-+ } t_1 \; t_2 \; t_3\texttt{)} :: (Trnsfr[\![t_3, \textbf{nil}, d_1]\!]\,(penv, denv) @ Tr[\![r]\!]\,(penv, denv)) \\
\qquad \text{if } penv(p_1) = \textbf{arith3-+}(d_1,(t_1,t_2,t_3)) \\
Tr[\![\texttt{(throw } v_h \; d_1\texttt{)} :: r]\!]\,(penv, denv) = \\
\qquad (Trnsfr[\![v_{l1}, fl_1, d_1]\!]\,(penv, denv) @ \cdots @ Trnsfr[\![v_{ln}, fl_n, d_1]\!]\,(penv, denv)) @ \\
\qquad Tr[\![r]\!]\,(penv, denv) \\
\qquad \text{where } Decomp(v_h) = \{(v_{l1}, fl_1), \ldots, (v_{ln}, fl_n)\} \\
\\
Trnsfr[\![v_l, \textbf{nil}, d_1]\!]\,(penv, denv) = [\texttt{(assign } v_l \; t_1\texttt{)}, \texttt{(get-rqst-mbox-on } t_1 \; t_2\texttt{)}] \\
\qquad \text{if } denv(d_1) = (p_1, 1) \text{ and } penv(p_1) = \textbf{req-send}(t_1,t_2,t_3) \\
Trnsfr[\![v_l, fl, d_1]\!]\,(penv, denv) = [\texttt{(setarg-remote } v_l \; t_2 \; fl\texttt{)}] \\
\qquad \text{if } denv(d_1) = (p_1, 2) \text{ and } penv(p_1) = \textbf{req-send}(t_1,t_2,t_3) \\
Trnsfr[\![v_l, fl, d_1]\!]\,(penv, denv) = Trnsfr[\![v_l, (i-1) :: fl, d_2]\!]\,(penv, denv) \\
\qquad \text{if } denv(d_1) = (p_1, i) \text{ and } penv(p_1) = \textbf{make-tuple}(d_2) \\
Trnsfr[\![v_l, \textbf{nil}, d_1]\!]\,(penv, denv) = [\texttt{(assign } v_l \; t_i\texttt{)}] \\
\qquad \text{if } denv(d_1) = (p_1, i) \text{ and } penv(p_1) = \textbf{arith3-+}(d_3,(t_1,t_2,t_3)) \;\; (\text{for } 1 \le i \le 2)
\end{array}
$$

Figure 2: Translation Function from Plan-Do Style Code into Lower-Level Code for Pipelined Sends.

Trnsfr realizes data transfer to the destination d in the following way :

- If d is for a target of request message send, the value is simply assigned to the temporary variables for the target, but, immediately after that, it allocates a message box for the pipelined send.
- If d is for an argument of a message, it generates a "remote write" instruction for the pipelined send.
- If d is for a tuple construction, it eagerly transfers the value to the more specific field of d by recursively calling *Trnsfr*.
- If d is just for an arithmetic operation, the value is simply assigned to the corresponding temporary variable.

Figure 3 shows the result of the conversion from the high-level code into lower-level code for `[obj1 <= [(+ i 1) (+ j 2)]]`. The high-level code is also shown on the left

```
 Plan-Do Style Code                       Lower-Level Code
(newplan (p1 d1 d2) (request-send))
(throw obj1 d1)                          (assign obj1-0 t1)
                                         (get-rqst-mbox-on t1 t2)  [*1]
(newplan (p2 d3 d4) (make-tuple d2))
(newplan (p3 d5 d6) (arith3-+ d3))
(throw i d5)                             (assign i-0 t4)
(throw 1 d6)                             (assign 1 t5)
(do p3)                                  (arith3-+ t4 t5 t6)
                                         (setarg-remote t6 t2 (0)) [*2]
(newplan (p4 d7 d8) (arith3-+ d4))
(throw j d7)                             (assign j-0 t7)
(throw 2 d8)                             (assign 2 t8)
(do p4)                                  (arith3-+ t7 t8 t9)
                                         (setarg-remote t9 t2 (1)) [*3]
(do p2)
(do p1)                                  (get-read-pointer t2 t3)  [*4]
                                         (request-send t3 t1)      [*5]
```

`[*1]` allocate a message box t2 on the node of t1, `[*2]` pipelined remote write of i+1, `[*3]` pipelined remote write of j+2, `[*4]` complete the initialization of the message box, `[*5]` send the message box address.

Figure 3: Generated Lower Level Code for `[obj1 <= [(+ i 1) (+ j 2)]]`

hand side. Eager data transfers become manifest at `(do p3)` and `(do p4)`, where the elements of the tuple are eagerly sent to the pre-allocated message box on a remote node. Note that `(do p2)` does nothing because the tuple elements are already sent eagerly.

4 Performance Measurements

In this section, we demonstrate the effect of eager data transfer, especially pipelined sends, via performance measurements on a COOP language system ABCL/EM-4[8, 9].

The source language of ABCL/EM-4 is ABCL/ST, a statically-typed ABCL language. An ABCL/ST compiler for EM-4 has already been developed [7]. The compiler adopts the following language hierarchy for each level of the abstraction of 'directly communicating thread': (1) the source language, (2) high-level language (embodying the semantics of the source language in terms of sequential execution of a single thread of control), (3) middle-level language (introducing pointers for the purpose of implementation), (4) low-level language (introducing the notion of data size and memory addresses), (5) I-language (for optimization based on flow analysis), and (6) assembly language. The plan-do style is mainly used for the high-level code to provide dataflow information without loss of machine independence of the high-level code.

The compiler generates assembly code for a fine-grained *hybrid* parallel architecture EM-4[4, 3], which was developed and built at Electrotechnical Laboratories. EM-4 consists of 80 processing elements and runs at 12.5 MHz clock speed, and facilitates a fine-grained communication mechanism; for example, it provides a two-words size packet-output instruction which directly sends data from registers into its fast omega network. EM-4 hardware also provides basic scheduling mechanism for multiple threads, coupled with its data-driven (packet-driven) feature. The term "hybrid" indicates the combination of

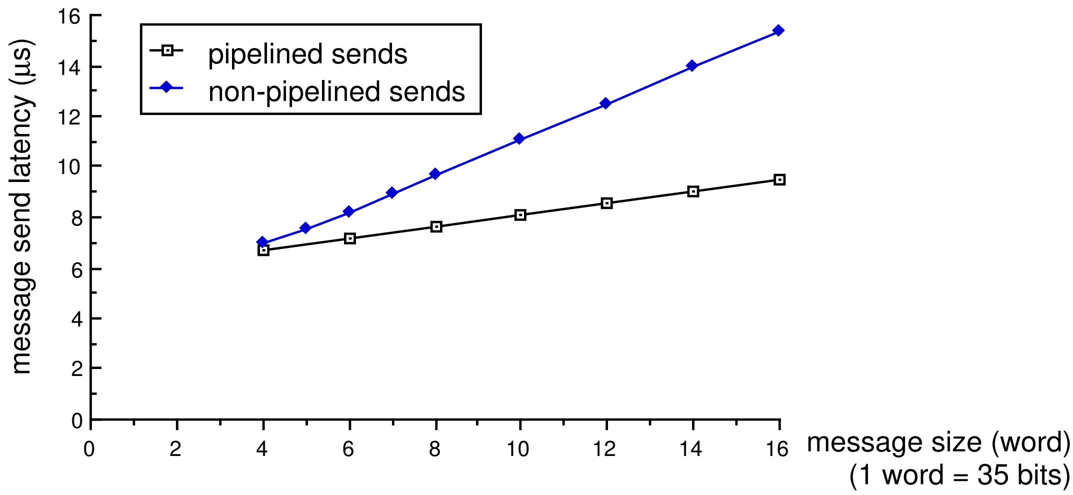

Figure 4: Latency Comparison Between Pipelined/Non-Pipelined Sends

control-flow architecture and the data-driven architecture.

In this measurement, we compare the two implementation schemes of remote message passing: (1) the pipelined sends (see Section 2.1) and (2) the non-pipelined sends, in which the communication is not initiated until all the message arguments are evaluated.

In order to measure the latency of remote message sends, our benchmark program creates objects along a closed Hamilton path among the 80 nodes of EM-4, and sends messages through the path. When an object O_i is activated by a message M_i from O_{i-1}, O_i sends M_{i+1} to O_{i+1} at the adjacent node along the path.

We measured the average activation interval of adjacent objects O_i and O_{i+1} for various sizes of messages. Figure 4 shows the results of the benchmark. As we can see, the pipelined sends are always superior to the non-pipelined sends. The main reason is because the non-pipelined sends require additional memory accesses to buffer the message arguments at the sender node; this buffering is not necessary for the pipelined sends, where the evaluated value on a register is sent directly into the network.

The latency given in Figure 4 includes not only the communication latency but also the execution time of computation performed by objects. Nevertheless, the minimum latency is approximately 6.7 μs (83 clock cycles) for 4-word message size in pipelined sends, while the difference between the latency in non-pipelined sends and the latency in pipelined sends for 16-word message size is approximately 5.8 μs (73 clock cycles). These values show that the impact of pipelined sends becomes relatively significant on architectures where the remote communication is very fast.

5 Discussion

5.1 Moderate Eagerness

The idea of eager data transfer originates from the dataflow execution model. In the dataflow execution model (the *eager evaluation* model), a large number of fine-grained concurrency can be exploited, ideally resulting in perfect speed up of the execution. In practice, however, the explosion of concurrency frequently causes problems, such as the resource exhaustion and the network contention.

In the thread-based execution model, concurrency can be easily bounded by the number of threads. Moreover, each thread execution can be efficiently performed by the use of pipelines and registers. Recent approaches, such as TAM[2], attempt to take advantage of this fact by resorting to compiler-assisted, highly-efficient thread-based execution model, even for languages which have fine-grained lenient semantics.

Our approach introduces eagerness in the compilation of communication in the directly-communicating threads model to exploit the fine-grained capabilities of recent architectures. We note, however, this does not cause the same problem (explosion of concurrency) as *eager evaluation*: in our approach, the eagerness appears only in the form of eager data transfer, which does not include "fire" as in dataflow execution (which consists of eager data transfer + matching + fire). The eagerness in our approach is still well moderated by the thread-based execution model.

5.2 Scheduling

Our abstract machine model employs a simple hierarchy of scheduling, namely (1) scheduling of threads and (2) scheduling of instructions. In ABCL/EM-4, scheduling of threads is basically performed by the hardware at runtime, because EM-4 quite efficiently supports the scheduling of threads, including creation and termination of threads, with its data-driven features. Instruction scheduling within a thread is the responsibility of the compiler, and the plan-do style is employed for this purpose. But plan-do style actually plays a more important role in instruction conversion between different abstraction levels.

5.3 Combining with Other Optimization Techniques

When an optimization based on dataflow information is already required at the higher-level, generation of plan-do style code described in Section 3.1 is not necessary. In such cases, only the conversion into the lower-level code described in Section 3.2 needs to be applied for realizing eager data transfer.

For example, in array computations, [1] generates optimized code of 'directly communicating threads,' where the optimization is based on dataflow information (rather than data dependency). In non-strict computations, [5] generates optimized code of 'directly communicating threads,' by partitioning the dataflow graph. In both cases, by keeping the dataflow information in the threads, our compilation scheme described in Section 3.2 would be extended to serve as a backend which supports their communication, when compiling into further fine-grained communication is desired for architectures such as EM-4.

6 Conclusion

We have presented a novel compilation-based approach to realizing high-throughput low-latency communication for the execution models based on *directly communicating threads*. We employed a compilation-based approach rather than a library-based one, exploiting advanced implementation techniques for communication. In particular, we concentrated on *eager data transfer* and described its compilation framework. We demonstrated the effectiveness of eager data transfer when employed with compiled *pipelined sends*, via performance measurements on a fine-grained hybrid parallel architecture EM-4.

In order to realize eager data transfer, dataflow information is indispensable to code conversion. The compiler must provide additional dataflow information even at the levels where optimization techniques based on dataflow information are not applied. This requirement causes a difficulty in combining several compilation phases into one pass. Our *plan-do style* intermediate code solves this problem by representing the combination of control-flow information and data-flow information in a stream form.

Acknowledgements

We are grateful to Drs. Yoshinori Yamaguchi, Mitsuhisa Sato, Yuetsu Kodama at ETL and Dr. Shuichi Sakai at RWCP for allowing us to use EM-4 and sparing time for technical discussions. We also thank Prof. Kei Hiraki at the Univ. of Tokyo for his helpful comments. Hidehiko Masuhara's comments on an early draft are appreciated. Masahiro Yasugi is supported in part by a JSPS postdoctoral Fellowship.

References

[1] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," *Proc. of PLDI'93*, pp. 126–138, 1993.

[2] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proc. of ASPLOS IV*, pp. 166–175, April 1991.

[3] Y. Kodama, S. Sakai, and Y. Yamaguchi, "A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation," *Proc. of InfoJapan '90*, pp. 291–298, 1990.

[4] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An Architecture of a Dataflow Single Chip Processor," *Proc. of ISCA'89*, pp. 46–53, June 1989.

[5] K. R. Traub, D. E. Culler, and K. E. Schauser, "Global Analysis for Partitioning Non-Strict Programs into Sequential Threads," *Proc. of LFP'92*, June 1992.

[6] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," *Proc. of ISCA'92*, pp. 256–266, May 1992.

[7] M. Yasugi, "A Concurrent Object-Oriented Programming Language System for Highly Parallel Data-Driven Computers and its Applications," Technical Report 94-7e, Department of Information Science, University of Tokyo, Apr. 1994. (Doctoral Thesis).

[8] M. Yasugi, S. Matsuoka, and A. Yonezawa, "ABCL/onEM-4: A New Software/Hardware Architecture for Object-Oriented Concurrent Computing on an Extended Dataflow Supercomputer," *Proc. of ICS'92*, pp. 93–103, July 1992.

[9] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura, Implementing Concurrent Object-Oriented Languages on Multicomputers, *IEEE Parallel & Distributed Technology*, Vol. 1(2), pp. 49–61, May 1993.

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