Implementing a Non-strict Functional Programming Language on a Threaded Architecture

Shigeru Kusakabe†, Kentaro Inenaga†, Makoto Amamiya†,
Xinan Tang‡, Andres Marquez‡, and Guang R. Gao‡

†Dept. of Intelligent Systems, Kyushu University,
‡EE & CE Dept. University of Delaware
E-mail: kusakabe@is.kyushu-u.ac.jp

Abstract. The combination of a language with fine-grain implicit parallelism and a dataflow evaluation scheme is suitable for high-level programming on massively parallel architectures. We are developing a compiler of V, a non-strict functional programming language, for EARTH(Efficient Architecture for Running THreads). Our compiler generates codes in Threaded-C, which is a lower-level programming language for EARTH. We have developed translation rules, and integrated them into the compiler. Since overhead caused by fine-grain processing may degrade performance for programs with little parallelism, we have adopted a thread merging rule. The preliminary performance results are encouraging. Although further improvement is required for non-strict data-structures, some codes generated from V programs by our compiler achieved comparable performance with the performance of hand-written Threaded-C codes.

1 Introduction

Many fine-grain multithreaded architectures have been proposed as promising multiprocessor architectures because of their ability to tolerate communication and synchronization latencies inherent in massively parallel machines[1][3][9][12][14][18]. By providing a lot of threads and supporting fast switches among threads, multithreaded architectures can hide communication and synchronization latencies. Control of execution is switched to a new thread when a long-latency operation is encountered. Many explicit parallel languages have been proposed for fine-grain multithreaded programming[7][8][17][19]. Overlapping computation and communication is programmer's task when using this kind of explicit languages. Writing explicit parallel programs on parallel machines is still a skilled job.

We are developing a high-level parallel programming language, called "V," which would minimize the difficulties in writing massively parallel programs[15]. In order to provide a high-level abstraction, the language is a non-strict functional programming language with implicit parallelism. There is no anti-dependence in V programs and it is easy to extract parallelism at various levels including fine-grain parallelism from V programs. The underlying computation model is an optimized dataflow computation model, Datarol[4]. The combination of a functional language with implicit parallelism and a fine-grain multithread evaluation scheme is suitable for massively parallel computation. The languages abstracts the timing problems in writing massively parallel programs, while fine-grain multithread evaluation supports efficient execution of a large number of fine-grain processes for implicit parallel functional programs in a highly concurrent way.

This approach can exploit irregular and dynamic parallelism, thus support efficient execution of "non-optimal" codes. Since our language does not support explicit descriptions for parallel execution and data-mapping, it is necessary for the compiler to automatically extract parallel codes of optimal grain size, partition data-structures and map them to each processing node. However, this is not an easy task, and the compiler may generate non-optimal codes of non-uniform grain size and ill-mapped data-structures. The multithreaded architectures with the ability to tolerate computation and synchronization latencies alleviate the problems of non-optimal code.

In this paper, we discuss implementation issues of V on a multithreaded architecture EARTH (Efficient Architecture for Running THreads). In order to show the feasibility of our language, we have implemented our language on commercially available machines such as Sequent Symmetry and Fujitsu AP1000[16]. However, since our basic execution model is a multithreaded execution model extended from Datarol model, multithreaded architectures which have special support for fine-grain parallel processing are preferable for our language. EARTH realizes efficient multithreading with off-the-shell microprocessor-based processing nodes[12]. We are implementing V on EARTH, while there are two explicit parallel languages for EARTH, EARTH-C[11] as a higher-level language and Threaded-C[21] as a lower-level language. The goal of this work is to show adequate compiler support makes our implicit parallel language as efficient as the explicit parallel languages on the multithreaded architecture. Our final aim is to realize a high-level programming environment on a high performance architecture.

In compilation, we use a virtual machine code, DVMC (Datarol Virtual Machine Code), as an intermediate code, and Threaded-C as a target code. Threaded-C is an explicit multi-threaded language, targeted for the EARTH model. We have developed translation rules, and integrated them into the compiler. Since overhead caused by fine-grain processing may degrade performance for programs with little parallelism, we have adopted a thread merging rule. The preliminary performance results on EARTH-MANNA[12][10] are encouraging. Although further improvement is required for non-strict data-structures, some codes generated from V programs by our compiler achieved comparable performance with the performance of hand-written Threaded-C codes.

This paper is organized as follows: section 2 and section 3 introduce our language V and EARTH respectively. Section 4 explains our compilation method. Section 5 discusses implementation issues of fine-grain data-structures on EARTH. Section 6 makes concluding remarks.

2 Language V

2.1 Language feature

Originally, V is a functional programming language Valid[2], developed at NTT ECL for colored token dataflow architectures as a part of the Dataflow Project. The fundamental design policies are:

- Recursive functional programming was adopted as the basic structure.
- A static binding rule was applied to variables.
- A static type system was adopted.
- Call-by-value was adopted as the fundamental computation rule.

Valid provided facilities for highly parallel list processing, and demonstrated the feasibility of functional programming and dataflow computation.

A program consists of a set of user-defined functions, and their invocations. A function instance is invoked using a function definition. The following is a form of function definition:

```
function ⟨function_definition_name⟩ (⟨formal_parameters⟩)
 return (⟨types_of_return_values⟩)
 = ⟨body_expression⟩ ;
```

Notation ⟨function_definition_name⟩ is an identifier for the function. Notation ⟨formal_parameters⟩ specifies the names and types of formal parameters, and ⟨types_of_return_values⟩ the types of return values. As an example, we consider the following program which calculates the summation from `low` to `high`:

```
function summ(low,high:integer) return(integer)
= if low=high then low
  else {let mid:integer=(low+high)/2 in summ(low,mid)+summ(mid+1,high)};
```

This function has two integer inputs `low` and `high`, and one integer return value. Within the function body, two recursive calls may be invoked.

2.2 Intermediate code

There is no anti-dependence in V programs and it is easy to extract parallelism at various levels including fine-grain parallelism from V programs. Parallelism is two-fold: function applications and subexpressions. Subexpressions executable in parallel are compiled to independent threads.

As an intermediate code, our compiler uses DVMC, which is a dataflow virtual machine code based on the Datarol model. The idea of Datarol is to remove redundant dataflow by introducing registers and a by-reference data access mechanism. A Datarol program (DVMC) is a multithread control-flow program, which reflects the dataflow inherent in the source program. While Datarol reflects the underlying data-flow structure in the given program, it eliminates the redundant data-flow operations such as switch and gate controls, and also eliminates the operand matching overhead. In Datarol, a function (procedure) consists of multiple threads, and an instance frame (virtual register file) is allocated in function invocation. The threads within the function share the context on the frame. While a data-driven mechanism is used to activate threads, result data and operand data are not passed as explicit data tokens, but managed on the instance frame.

Lenient evaluation (non-strict and eager dataflow evaluation) can maximally exploit fine-grain parallelism and realize flexible execution order. Argument expressions for a function invocation can be evaluated in parallel, and the callee computation can proceed if either of argument value is prepared. the "call f r" creates an instance of the function f, and stores the pointer to the instance in r. The "rins" releases the current instance. The "link r v *slot*" sends the value of v as the *slot*-th parameter data to the callee instance specified by r. The "rlink r *cont slot*" sends the continuation, *cont* for the returned value and the continuation threads in the callee instance, as the *slot*-th parameter data to the callee instance specified by r. The "receive *slot* v" receives the data transferred from the caller instance through the *slot*-th slot, and stores the data into v. The " return rp v" triggers the continuation threads specified by the rlink operation after setting the value of v to the content of rp.

Fig.1 shows DVMC for the summation program, generated by a straightforward compilation method. In DVMC, execution control among threads proceeds along continuation points. In the figure, a solid box represents a thread, an arrow a continuation arc, and a wavy arrow an implicit dependence.

In the figure, three `RECEIVE`s at the top are nodes to receive input parameters, two integers `low` and `high`, and one continuation point to return the result value. Within the function body, two shaded parts correspond to two recursive calls. A `LINK` is a node to send parameter value, and an `RLINK` to send a continuation point to receive a result value.

3 EARTH

EARTH architecture is a multiprocessor architecture designed for the efficient parallel execution of both numerical and non-numerical programs. The basic EARTH design begins with a conventional processor, and adds the minimal external hardware necessary for efficient support of multithreaded programs.

In the EARTH model, a multiprocessor consists of multiple EARTH nodes and an interconnection network. The EARTH node architecture is derived from the McGill Data Flow Architecture, which was based on the idea that synchronization of operations and the execution of the operations themselves could be performed more efficiently in separate units rather than in the same processor. Each EARTH node consists of an Execution Unit (EU) and a Synchronization Unit (SU), linked together by buffers. The SU and EU share a local memory, which is part of a distributed shared memory architecture in which the aggregate of the local memories of all the nodes represents a global memory address space.

The EU processes instructions in an active thread, where an active thread is initiated for execution when the EU fetches its thread id from the ready queue.

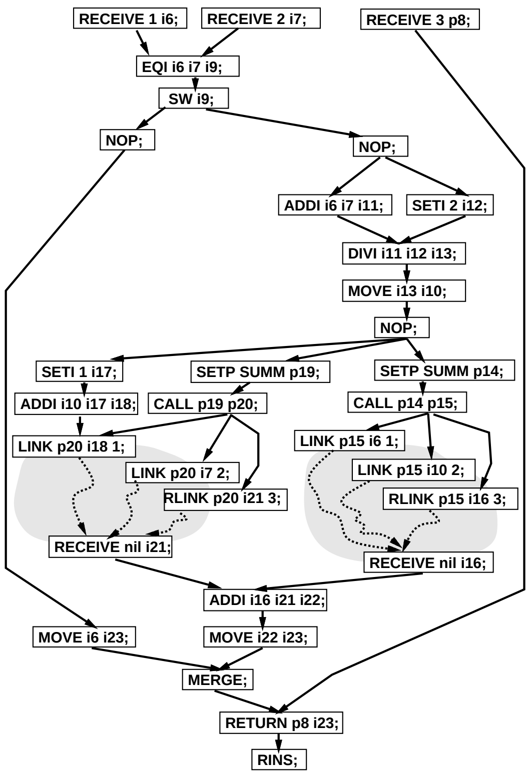

Fig. 1. Fine-grain abstract machine code of `summ`.

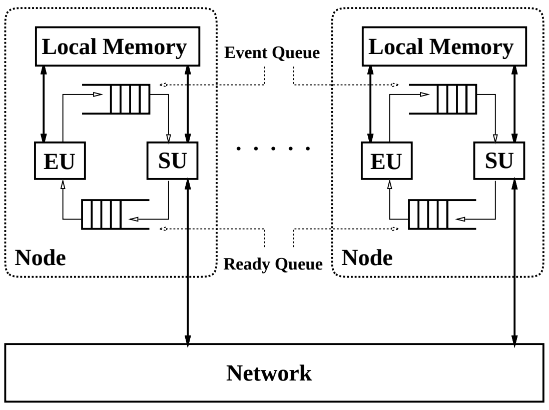

Fig. 2. EARTH architecture

The EU executes a thread to completion before moving to another thread. It interacts with the SU and the network by placing messages in the event queue. The SU fetches these messages, plus messages coming from remote processors via the network. The SU responds to remote synchronization commands and requests for data, and also determines which threads are to be run and adds their thread ids to the ready queue.

The EARTH programming model is implemented as an extension to the C language. The resulting C dialect, EARTH Threaded-C, is an explicitly parallel language that allows the programmer to directly specify the partitioning into threads and the EARTH operations[21]. The higher-level C dialect for EARTH is EARTH-C[11], which has simple extensions to express control parallelism, data locality, and collective communication. The EARTH-C compiler generates the Threaded-C code as output.

For preliminary performance evaluation, we use an implementation of EARTH on MANNA (EARTH-MANNA). The MANNA (Massively parallel Architecture for Non-numerical and Numerical Applications) architecture was developed at GMD FIRST in Berlin, Germany. Each node of a MANNA multiprocessor consists of two Intel i860 XP RISC processors clocked at 50 MHz, 32 MB of dynamic RAM and a bidirectional network interface. The link interface is capable of transferring 50 MB/s in each direction simultaneously, for a total bandwidth of 100 MB/s per node. The network is based on 16 x 16 crossbar chips which support the full 50 MB/s link speed.

4 Compilation

In this section, we explain thread level compilation rules, and discuss effectiveness of static thread scheduling.

4.1 Abstract machine

Fig.3 shows the schematic view of our abstract runtime model. We employ a two-level scheduling: instance frame level scheduling, and thread level scheduling within an instance frame. Instance frames hold the local variables and synchronization variables. Threads within a function share the context on an instance frame for the function. A thread is a schedulable unit, and each thread has a synchronization counter variable whose initial value is determined at compile time. When a thread is triggered, the counter is decremented. If it reaches zero, the thread becomes ready and enqueued into the thread queue. Operations with unpredictably long latency are realized as a split-phase transaction, in which one thread initiates the operation, while the operation that uses the returned value is in another thread and triggered by the returned value.

The thread queue is provided in order to dynamically schedule intra-instance threads. If the execution of the current thread terminates, the next ready thread is dequeued from the thread queue for the next execution. If the thread queue is exhausted before the instance terminates, the instance becomes a "suspended instance" and is stored into the idle pool before the execution switches to another frame. If the instance terminates, the runtime system releases the instance frame. The runtime system manages runnable instance frames by using the instance queue. The runtime system picks up one of the runnable frames and passes the execution control to the corresponding code to activate the instance frame. When a frame in the idle pool (a suspended instance frame) receives data from

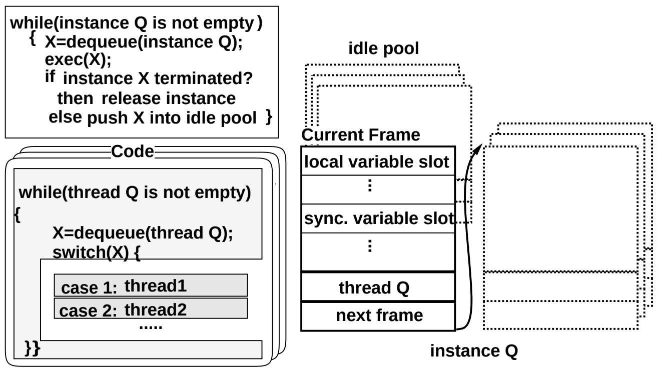

Fig. 3. Overview of abstract machine.

another active frame, the corresponding thread in the suspended frame is triggered. If the thread becomes ready and is enqueued into the thread queue, the instance frame also becomes ready and is moved to the instance queue.

4.2 Compiling to Threaded-C

The compiler must generate an explicit threaded code from an implicit parallel V program. In compilation, we use a virtual machine code DVMC as an intermediate code, and Threaded-C as a target code.

We explain the compilation rules for DVMC to generate Threaded-C code. As shown in the previous sections, both Datarol virtual machine and EARTH architecture support threaded execution. However, the two-level scheduling in DVMC, instance frame level scheduling and thread level scheduling, is merged and mapped to the ready queue in EARTH.

Table 1 shows basic translation rules. Basically, since a function instance in DVMC is a parallel entity, a function in DVMC is translated to a THREADED function in Threaded-C. By using a TOKEN instruction for a THREADED function, the function instance is forked in Threaded-C. Both DVMC functions and THREADED functions use heap frame as their activation record. There are two types of function calls in DVMC: strict call and non-strict call. As a usual convention, a strict call does not start until all the arguments become available. In con-

DVMC		**Threaded-C**
function	⇒	THREADED function
thread	⇒	THREAD
basic instruction	⇒	basic instruction
strict call	⇒	TOKEN instruction
non-strict call	⇒	TOKEN (with I-structure)

Table 1. Translation rules

trast, a non-strict call starts its execution before the arguments become available. This execution style overlaps computations among caller and callee side, maximally exploits fine-grain parallelism, and realizes flexible execution order, since computation can be triggered by a subset of the arguments. Although function call in Threaded-C is basically strict, by using I-structure library[5], we can realize non-strict call in Threaded-C.

In Threaded-C, a thread is enclosed by `THREAD_id` and `END_THREAD()`. In DVMC, a thread is started with a thread label and ended with ! mark. Each instruction has a continuation tag, showing which threads are waiting for this instruction to finish. In contrast to Threaded-C, the number of the synchronization signals that a thread waits for is not explicitly coded in the instruction sequence in DVMC. It relies on the compiler to scan the sequence to compute synchronization information for each thread. On contrary, Threaded-C needs the programmer to explicitly specify *sync* signals that a thread waits for by the INIT_SYNC primitive. In Threaded-C, each thread is associated with a *sync* slot, which is a quadruple: *(slot-number, sync-count, reset-count, thread-number)*. The *reset-count* is necessary when *sync-count* becomes zero and the thread needs to restart, for example a thread within a loop. The *sync* slots need to be initialized in advance. Thus, the first few instructions in any Threaded-C function are usually INIT_SYNC primitives, assigning *sync* slots to threads and setting *sync* counts accordingly. (DVMC and Threaded-C versions of Fibonacci program `fib` are shown in Fig.7 in appendix)

4.3 Static scheduling

Although EARTH supports thread level execution, excessively fine-grained threads may incur heavy overhead. Compile-time scheduling is effective to reduce run-time cost of scheduling fine-grain activities. In scheduling a non-strict program, cares must be taken for achieving high performance while keeping non-strict semantics of the program.

Our scheduling method pays attention not to violate non-strict semantics. Although another scheduling is possible for software implemented DVMC system[13], in which blocking threads can be realized by software support, the compiler for EARTH, in which a thread is no-preemptive, adopt a compile-time scheduling similar to separation constraint partitioning[20]. The basic rule for the thread scheduling is that two nodes must reside in different threads if there exists any indirect dependency between them. Indirect dependences, such as a certain indirect dependence due to access to asynchronous data and a potential indirect dependence due to function call, may require dynamic scheduling. In our scheduling, a merging policy is introduced to make more functions fit conventional invocation style.

Fine-grain codes such as shown in Fig.1 will incur heavy overhead. Our static scheduler can translate such a fine-grain code into a coarse grain code as shown in Fig.4. In the figure, shaded parts are two recursive calls. As a result of static thread merge, operations concerning each function call are merged into a single thread, and the function is invoked as a strict function. This code can be translated to a Threaded-C program, and can be executed efficiently on EARTH.

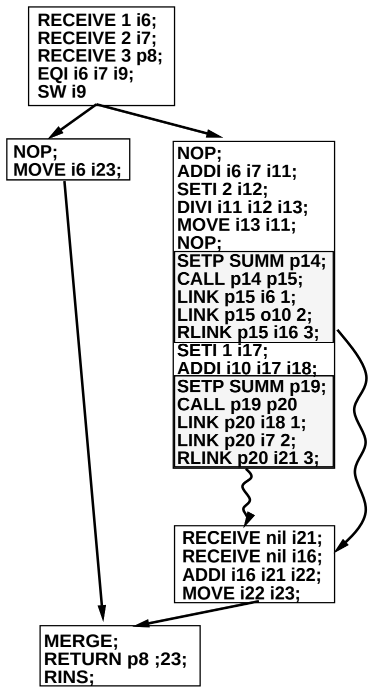

Fig. 4. Code after static thread merging.

Following is a result of preliminary performance evaluation on EARTH-MANNA using Fibonacci program. The compiled version is a Threaded-C program generated from V program by the compiler. The hand-coded version is a Threaded-C program written by a programmer using the same algorithm as the V source program. As shown in the table, the automatically generated compiled version runs as fast as the hand-coded version on EARTH-MANNA.

	n		
	16	24	32
compiled[ms]	2	59	1984
hand-coded[ms]	2	56	1970

Table 2. Elapsed time of fib(n) on EARTH-MANNA

5 Fine-grain parallel data-structures

Non-strict data-structures alleviate the difficulty of programming for complex data-structures[6]. Due to non-strictness, programmers can concentrate on the essential dependencies in the problems without writing explicit synchronizations between producers and consumers of data-structure elements. In addition to this advantage of abstracting timing problems, non-strict data-structures have another advantage to communication overhead, which is one of the problems in parallel processing. An eager evaluation scheme of non-strict data-structures increases potential parallelism during program execution. This evaluation scheme is effective for hiding communication overhead, by switching multiple contexts while waiting for the result of communication results.

Partitioning and distribution of large data-structures have a large impact on the parallel processing performance. Although the owner computes rule is known to be effective for SPMD style programming to some extent, such a monolithic rule is not suitable for MIMD style programming, whose computation and communication pattern may complex and unpredictable. The combination of non-strict data-structures and the eager evaluation can eliminate descriptions of explicit partitioning and distribution of large data-structures, as well as explicit synchronization in MIMD style programming. The multithreaded execution with the ability to tolerate computation and synchronization latencies alleviate the problems of non-optimal code such as non-uniform grain size and ill-mapped data-structures. However, non-strict data-structures require frequent dynamic scheduling at a fine-grain level during program execution. Fine-grain dynamic scheduling causes heavy overhead, which offsets the gain of latency hiding. In this section, we discuss on the implementation issues of non-strict data-structures.

5.1 Implementation model

An array is generated by a bulk operator, `mkarray`, in V. In creating a non-strict array, the allocation of the array block and the computations to fill the elements are separated. In a dataflow computation scheme, a `mkarray` immediately returns an array block (pointer) and calls the filling functions in parallel. Fig.5 outlines the `mkarray` computation model on a distributed memory machine. In the figure, the `mkarray` node is an instance representing an array, a `mksubarray` node is an instance representing a subarray block on a node, and `pf`s are instances of filling functions for array elements. On distributed-memory machines, an array is often distributed over the processor nodes, and filling functions are scattered among the processor nodes. We map the filling functions using the owner computes rule as a basic rule. Although the `pf`s may activated in parallel, they may be scheduled according to data dependencies.

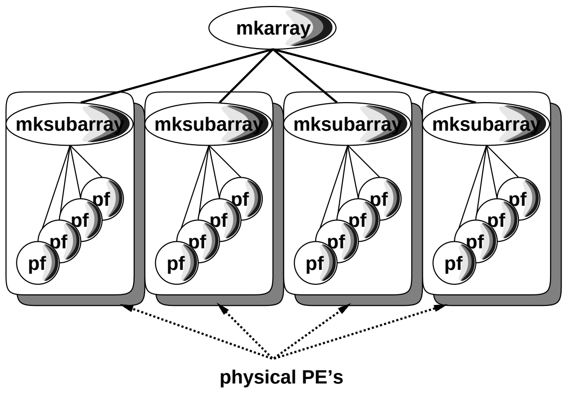

Fig. 5. Schematic view of a `mkarray` computation model on a distributed memory machine.

Array descriptors support array distribution and realize global address access for distributed arrays. Array descriptors consist of a body, a decomposition table, and a mapping table, in order to manage the top-level information, the decomposition scheme, and the mapping scheme, respectively. The body has a total size, the number of dimension, the number of processors, and various flags. The decomposition table has the lower and the upper bounds of each dimension, and some parameters to specify the decomposition scheme: the number of logical processors, the block size, and the base point. The mapping table has correspondences between the logical and physical processors for the sub-arrays, the top of the local address for the block.

5.2 Discussion

According to this model, we implemented arrays in V. We also measured preliminary performance on EARTH-MANNA using matrix multiplication program `matmul`.

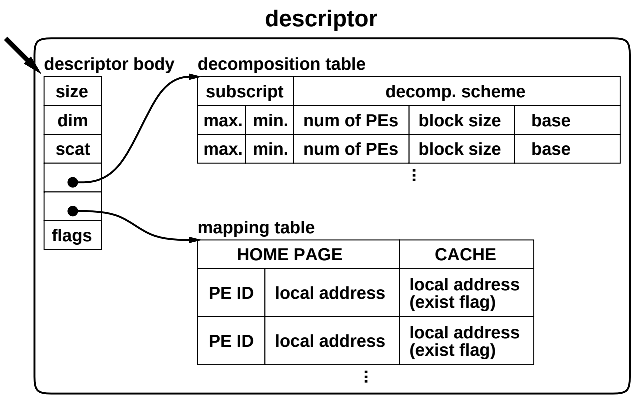

Fig. 6. Array descriptor

matrix size	10	20	30	40	50
elapsed time[sec]	$3.14*10^{-2}$	$4.10*10^{-1}$	3.27	10.0	52.1

Table 3. Elapsed time of `matmul` on EARTH-MANNA

This result is very slow. Although EARTH-MANNA has the external hardware necessary for efficient support of multithreaded execution, naive implementation of non-strict data-structures incurs heavy overhead. Arrays in V are non-strict, and support element-by-element synchronization. Ideally, fine-grain computation and communication are pipelined and overlapped in a dataflow computation scheme, without exposing the synchronization and communication overhead. However, the overheads of fine-grain processing degrade the performance.

Although our language is a fine-grain non-strict dataflow language and our implementation scheme employs a multithread execution model, access to a data-structure element is performed by means of pointers to heap areas. Thus, many optimization techniques proposed for conventional languages are also applicable to our implementation of fine-grain parallel data-structures. In order to reduce the overhead caused by the frequent fine-grain data access, we are considering to incorporate following optimization techniques:

- caching mechanism for fine-grain data accesses, and
- grouping mechanism for fine-grain data accesses.

The second technique is to transform non-strict access to data-structure into scheduled strict access, by data dependency analysis between producers and consumers at compilation phase. In this technique, as many as possible non-strict accesses could be transformed to strict accesses, while the parallelism would fall victim. So that, the compiler has the trade off between non-strictness with high parallelism and strictness with low parallelism. During the optimization process, following issues are considered: Fine-grain parallel data-structures of our language allow element-by-element access. This feature is one of the key points in order to write non-strict programs as well as parallel programs. Optimizations involving indiscreet grouping may reduce effective parallelism, and may lead to a deadlock at runtime in the worst case.

6 Concluding remarks

We discussed implementation issues of V, a non-strict functional programming language, on EARTH. Functional languages are attractive for writing parallel programs due to their expressive power and clean semantics. The goal is to implement V efficiently on EARTH, realizing a high-level programming language on a high performance architecture. The combination of a language with fine-grain parallelism and a dataflow evaluation scheme is suitable for high-level programming for massively parallel computation. In compiling V to EARTH, we use a virtual machine code DVMC as an intermediate code, and Threaded-C as target code. Threaded-C is an explicit multi-threaded language, targeted for the EARTH model. We executed sample programs on EARTH-MANNA. The preliminary performance results are encouraging. Although further improvement is required for non-strict data-structures, the codes generated from V programs by our compiler achieved comparable performance with the performance of hand-written Threaded-C codes.

References

1. Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 1–6, 1990.
2. M. Amamiya, R. Hasegawa, and S. Ono. Valid: A High-Level Functional Programming Language for Data Flow Machine. *Review of Electrical Communication Laboratories*, 32(5):793–802, 1984.
3. M. Amamiya, T. Kawano, H. Tomiyasu, and S. Kusakabe. A Practical Processor Design For Multithreading. In *Proc. of the Sixth Symposium on Frontiers of Masssively Parallel Computing*, pages 23–32, Annapolis, October 1996.
4. M. Amamiya and R. Taniguchi. Datarol:A Massively Parallel Architecture for Functional Languages. In *the second IEEE symposium on Parallel and Distributed Processing*, pages 726–735, December 1990.
5. Jose Nelson Amaral and Gunag R. Gao. Implementation of I-Structures as a Library of Functions in Portable Threaded-C. Technical report, University of Delaware, 1998.
6. Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Prog. Lang. and Sys.*, 11(4):598–632, October 1989.
7. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

8. David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.
9. Jack B. Dennis and Guang R. Gao. *Multithreaded computer architecture: A summary of the state of the art*, chapter Multithreaded Architectures: Principles, Projects, and Issues, pages 1–74. Kluwer academic, 1994.
10. W. K. Giloi. Towards the Next Generation Parallel Computers: MANNA and META. In *Proceedings of ZEUS '95*, June 1995.
11. Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH Multithreaded Architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.
12. Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A Study of the EARTH-MANNA Multithreaded System. *International Journal of Parallel Programming*, 24(4):319–348, August 1996.
13. K. Inenaga, S. Kusakabe, T. Morimoto, and M. Amamiya. Hybrid Approach for Non-strict Dataflow Program on Commodity Machine. In *International Symposium on High Performance Computiong (ISHPC)*, pages 243–254, November 1997.
14. Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X Parallel Computer: Architecture and Basic Performance. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.
15. S. Kusakabe and M. Amamiya. *Dataflow-based Language V*, chapter 3.3, pages 98–111. Ohmsha Ltd., 1995.
16. S. Kusakabe, T. Nagai, Y. Yamashita, R. Taniguchi, and M. Amamiya. A Dataflow Language with Object-based Extension and its Implementation on a Commercially Available Parallel Machine. In *Proc. of Int'l Conf. on Supercomputing'95*, pages 308–317, Barcelona, Spain, July 1995.
17. R. S. Nikhil. Cid: A Parallel, "Shared-Memory" C for Distributed-Memory Machines. *Lecture Notes in Computer Science*, 892:376–, 1995.
18. R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992. Also as CSG-memo-325-1, Massachusetts Institute of Technology, Laboratory for Computer Science.
19. Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi. EM-C: Programming with Explicit Parallelism and Locality for the EM-4 Multiprocessor. In Michel Cosnard, Guang R. Gao, and Gabriel M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 3–14, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Co.
20. K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation Constraint Partitioning — A New Algorithm for Partitioning Non-strict Programs into Sequential Threads. In *Proc. Principles of Programming Languages*, January 1995.
21. Xinan Tang, Oliver Maquelin, Gunag R. Gao, and Prasad Kakulavarapu. An Overview of the Portable Threaded-C Language. Technical report, McGill University, 1997.

Appendix

```
1:  RECEIVE 1 i5;
    RECEIVE 2 p6;
    SETI 1 i7;
    LTI i5 i7 i8;
    SWN i8 15;
    __SW_B 15 T;
    NOP ;
    SETI 1 i20;
    MOVE i20 i21 ->(3);
    __SW_E 15 T;
    __SW_B 15 E;
    NOP;
    SETI 1 i9;
    SUBI i5 i9 i10;
    SETN FIB n11;
    CALLS n11 f12 (i10) (i13)
                      -> (2);
    SETI 2 i14;
    SUBI i5 i14 i15;
    SETN FIB n16;
    CALLS n16 f17 (i15) (i18)
                      -> (2);
  ! __SW_E 15 E;
2:  RECEIVE NIL i13;
    RECEIVE NIL i18;
    ADDI i13 i18 i19;
  ! MOVE i19 i21 -> (3);
3:  MERGE 2;
    RETURN p6 i21;
  ! RINS;
```

(a) DMVC

```
THREADED fib(SPTR done, long int i5,
                long int *GLOBAL p6)
{
  SLOT SYNC_SLOTS [2];
  long int i7,i8,i9,i10,i13,i14,
              i15,i18,i19,i20,i21;

  INIT_SYNC(0, 2, 2, 2);
  INIT_SYNC(1, 1, 1, 3);
  i7 = 2;
  i8 = i5 < i7;
  if (i8)
  {
    i20 = 1;
    i21 = i20;
    SYNC(1);
  }
  else {
    i9 = 1;
    i10 = i5 - i9;
    TOKEN(fib, SLOT_ADR(0), i10,
                  TO_GLOBAL(&i13));
    i14 = 2;
    i15 = i5 - i14;
    TOKEN(fib, SLOT_ADR(0), i15,
                  TO_GLOBAL(&i18));
    END_THREAD();
THREAD_2:
    i19 = i13 + i18;
    i21 = i19;
  }
  END_THREAD();
THREAD_3:
  DATA_RSYNC_L(i21, p6, done);
  END_FUNCTION();
}
```

(b) Threaded-C

Fig. 7. DVMC and Threaded-C code of function `fib`