J Supercomput (2009) 47: 228–252
DOI 10.1007/s11227-008-0246-5

A continuation-based noninterruptible multithreading processor architecture

Satoshi Amamiya · Makoto Amamiya · Ryuzo Hasegawa · Hiroshi Fujita

Published online: 6 November 2008

Abstract Current trend of research on multithreading processors is toward the chip multithreading (CMT), which exploits thread level parallelism (TLP) and improves performance of softwares built on traditional threading components, e.g., Pthread. There exist commercially available processors that support simultaneous multithreading (SMT) on multicore processors. But they are basically based on the conventional sequential execution model, and execute multiple threads in parallel under the control of OS that handles interruptions. Moreover, there exist few languages or programming techniques to utilize the multicore processors effectively.

We are taking another approach to develop a multithreading processor, which is dedicated to TLP. Our processor, named *Fuce*, is based on the continuation-based multithreading. A thread is defined as a block of sequentially ordered instructions which are executed without interruption. Every thread execution is triggered only by the event called continuation.

This paper first introduces the continuation-based multithread execution model and its processor architecture then gives multithreaded programming techniques and the continuation-based multithreading language system *CML*. Last, the performance of the Fuce processor is evaluated by means of the clock-level software simulation.

S. Amamiya · M. Amamiya (✉) · R. Hasegawa · H. Fujita
Department of Intelligent System, Kyushu University, 744 Moto-oka Nishui-ku, Fukuoka, 819-0395, Japan
e-mail: amamiya@al.is.kyushu-u.ac.jp

S. Amamiya
e-mail: roger@al.is.kyushu-u.ac.jp

R. Hasegawa
e-mail: hasegawa@ar.is.kyushu-u.ac.jp

H. Fujita
e-mail: fujita@ar.is.kyushu-u.ac.jp

Keywords Multithreading · Parallel processing · Thread level parallelism · Multithreaded programming · Processor architecture

1 Introduction

The requirement for utilizing information and communication facilities on the Internet such as the grid computing and ubiquitous computing environment has been highly increasing these days. In order to construct the required infrastructure for this requirement, it is indispensable to develop parallel/distributed computing techniques. The current trend of research on such processors is toward the chip multithreading (CMT) [1, 2], which aims to exploit thread level parallelism (TLP) and to improve performance of softwares built on traditional threading components, e.g., Pthread. CMT is principally realized by a straightforward extension of conventional symmetric multiprocessor (SMP) techniques. Therefore, even though the advances of semiconductor technologies enable the CMT, SMP-based CMT processors would have the limits to scalable multithread processing if they are built only on the traditional sequential-computation-based framework.

These computer architectures and softwares depend on von Neumann principle. Now is the time to reconsider parallel/distributed processing from the fundamental principles, and to develop a completely new architecture.

We are taking another approach to develop the new architecture called *Fuce* [3, 4] and its softwares based on a new principle: *noninterruptible thread* that runs through its code without being interrupted by others, and *continuation* that rules execution ordering and synchronization among multiple threads.

Our processor, named *Fuce*, is designed based on the continuation-based multithreading. *Fuce* is evolved from dataflow architectures (e.g. [5–8]) to make it more practical. A thread is defined as a block of instructions that are sequentially executed without being interrupted. Every execution of thread is triggered by events called *continuation* [8]. Executions among threads are partially ordered, and their order is specified by *continuation*, that is, the *continuation-based multithreaded program*. *Fuce* is another type of CMT processor that executes those multithreaded programs. It aims to fuse the intraprocessor computation and interprocessor communication. Internal computations and external communications are handled uniformly by the continuation-based event-driven multithreading.

This paper presents the continuation-based multithread execution model and *Fuce* processor architecture. Then the paper outlines multithreaded programming techniques in *Fuce*. Lastly, performance of the *Fuce* processor is evaluated with the clock-level software simulator developed in parallel to the hardware implementation on FPGA.

2 Background and aim of research

Two approaches are considered on the approach to multithreading. One is fine-grain multithreading evolved from the framework of dataflow architecture, and the other

is coarse-grain multithreading in the framework of conventional sequential processors. The dataflow-based approach aims to improve the fine-grained dataflow parallelism toward coarser-gained, so-called macro-dataflow, that augments dataflow with control-flow, and has evolved into multithreading. This approach was active in the 1980s and 1990s [5–8]. Unfortunately, this approach seems to have declined. There are several reasons for this: (1) Although a dataflow-based scheme has the potential to maximally exploit parallelism, it is inefficient for sequential programs on which traditional software is built. (2) Data-driven mechanisms are expensive to implement in hardware. (3) It is difficult to extract spatial and temporal locality in dataflow-based computation. (4) The dataflow-based architecture offers a poor recipe for developing a practical language system like the *C* language, and Operating System, e.g., multitasking, memory management, I/O handling, and so on.

For these reasons, the main stream of research on the parallel and distributed processing is on the basis of sequential-processing-based commodity machines, e.g., cluster machines and grid computing. In this stream, there are two approaches to multithreading: simultaneous multithreading (SMT) and chip multithreading (CMT). The SMT approach combines multithreading with superscalar techniques. Instructions are simultaneously issued from multiple threads. Therefore, SMT utilizes ILP techniques in order to exploit parallelism in instruction level, and enhances the sequential-based processors [9]. Recently, advanced versions of commodity processors have been commercialized [10, 11]. However, the problem of these SMT approach is that it is limited in exploiting parallelism even if it uses ILP because it is built on the sequential-computation-based model.

The CMT approach aims at a simpler and more effective realization of thread-level parallelism on a chip multiprocessor (CMP) [12]. The idea of this approach is very simple, in principle, in that it integrates the symmetric multiprocessor (SMP) into a CMP. CMT processors have been commercialized already [1, 2, 11]. But the CMT processors have the same problem as the SMT processors.

In contrast, our research on *Fuce* architecture aims to develop a scalable multithreading processor, revisiting dataflow-based computation model in spite of its decline. The new idea in this research is the *continuation*-based event-driven computation. Thread execution is triggered by *continuation* signals sent by other threads either external or internal, and executions among multiple threads are controlled by *continuation* events.

The objective of our research is to solve the problems in dataflow-based architecture mentioned above. For this purpose, we are taking a comprehensive approach to continuation-based event-driven multithreading, from the programming model, language system, operating system to the processor hardware implementation.

For instance, we are now developing the continuation-based multithreading language *CML* and an OS kernel which is built through the multithreaded programming. The *Fuce* processor hardware is designed to support the OS kernel mechanism.

3 *Fuce* execution model and processor architecture

3.1 Continuation-based multithread execution

The core concept of the *Fuce* thread execution model is the event-driven computation [8]. Every thread receives *continuation* signals from other threads and its execution is triggered on the receipt of all corresponding *continuation*'s. The thread is defined as a sequentially ordered block of more than one machine instruction executed exclusively without being interrupted. Note that this thread definition differs from the typical definition of nonblocking thread [13–15] or the definition of *block* in TRIPS [16].

The main goal of the *Fuce* processor is to fuse inner-computation and external communication by the notion of continuation-based multithreading. Not only user-level programs but also even OS kernel programs including external I/O-event handling codes are composed of sets of noninterruptible threads.

The *Fuce* program is written as a set of functions. Each function is programmed using threads, and the corresponding function instances are executed in its runtime environment. Information of function instances is stored in a specially devised high speed-memory called *Activation Control Memory* (ACM). Figure 1 depicts the structure of ACM. Its structure is similar to the paging system used in virtual memory in typical operating systems. Each page in ACM is associated with a function instance, and information of all the threads involved in the function is recorded in the ACM page.

The information for controlling thread execution: sync-count, fan-in, code-entry and lock-bit, are stored in ACM. Sync-count is the number of continuations a thread is currently waiting for. Initially, the sync-count is set to the fan-in value of the thread. Code-entry is a pointer to the entry address of the thread code. Each thread ID is given according to the recording order of threads in a page. Every thread is efficiently identified by its page number and page displacement in ACM. The lock-bit is used as a semaphore. The initial value of lock-bit is zero. The base-address is a pointer to the entry address of data area used by the function instance.

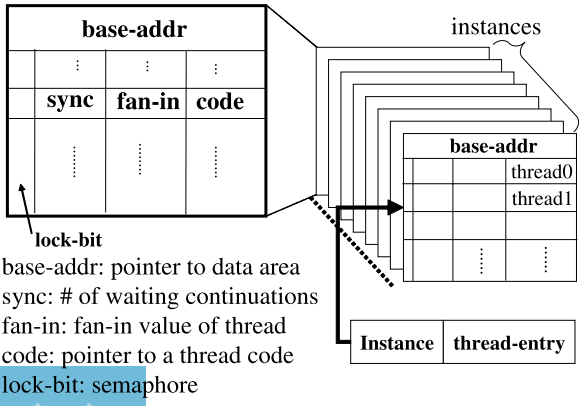


Fig. 1 The structure of ACM

3.2 *Fuce* processor

Figure 2 depicts an overview of the *Fuce* processor. In this processor, multiple thread execution cores are implemented in a chip. The key components of *Fuce* to support concurrent thread execution are the Thread Activation Controller (TAC), multiple Thread Execution Units (TEUs) and multiple register files.

3.2.1 Thread activation controller

Thread Activation Controller (TAC) controls all thread firings and mutual exclusions using ACM.

ACM is implemented in TAC as a high-speed memory using the same technology as cache memory. TAC handles the thread-control-related instructions such as `cont` and `newins` which are issued in the TEUs, and updates the states of ACM. A queue called ready-queue is implemented inside the TAC. TAC enqueues ready threads to the ready-queue. When one of TEUs finishes a thread execution, TAC allocates a new thread in the ready-queue to the free TEU.

In addition, the special high-speed memory called Bridge Registers (BR) is implemented inside TAC. BR is devised as the temporal data buffer used in the data passing between threads. As the data passing between threads occurs frequently in *Fuce*, it is desirable to make it as efficient as possible. BR has the similar paging structure to ACM, but its size is small enough.

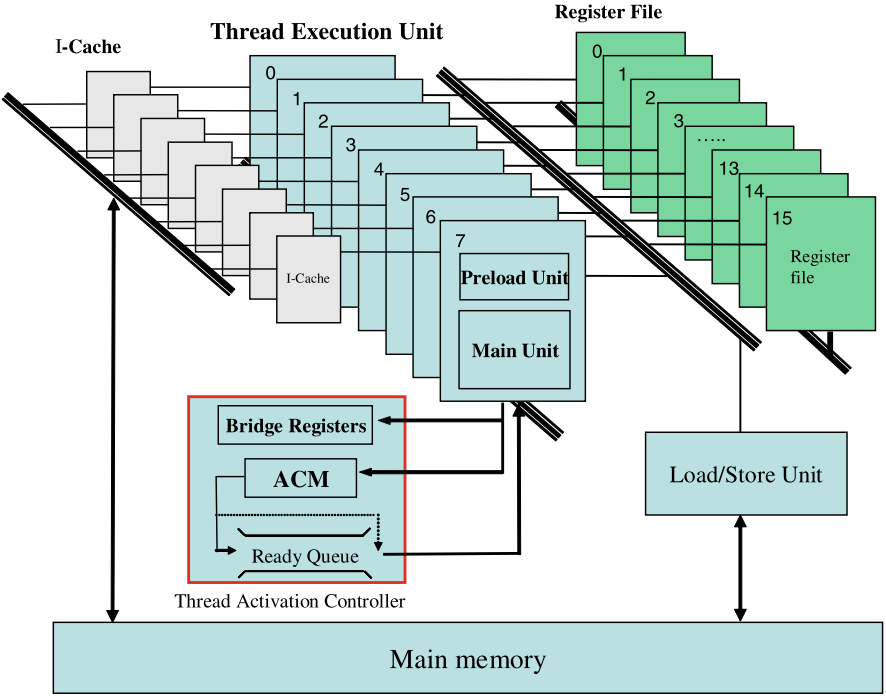


Fig. 2 The *Fuce* processor architecture

Table 1 Instruction set for *Fuce*

Thread instructions	Meaning
cont rs	continue to thread held in register rs
rcont rs	reset and continue to thread held in rs
delda rs	release data-area held in rs
delins rs	release ACM entry held in rs
end	terminate thread
newda rd, rs	get data-area of size held in rs and set to rd
newins rd, rs	get new ACM entry and set its page number to rd, then set to the entry the pointer to data-area held in rs
plend	end of preloading
setacm rs, rt, imm	set code address imm and fan-in value shown by rs to ACM entry shown by rt

3.2.2 Thread execution unit

Thread Execution Unit (TEU) executes instructions of thread. In the current implementation, the *Fuce* processor are equipped with eight TEUs to support concurrent execution of multiple threads. TEU consists of a Main unit (MU) and a Preloading unit (PLU). MU is a very simple 32-bit RISC processor and its internal architecture is quite similar to the MIPS processor. The instruction set of MU is also that of MIPS extended with thread-related instructions shown in Table 1. PLU is also a small RISC core, and it supports only load instructions. Note that the data-cache memory is not implemented because we consider there might not exist simple mechanisms to exploit data locality in the event-driven concurrent programs.

PLU and MU work with each other in pipeline fashion through a register file. PLU executes the fore-part of a thread, and MU executes the rest of the thread. Here, we assume that all load instructions are arranged into the fore-part of thread by the compiler or programmer.

4 Continuation-based multithread programming

4.1 Basic programming techniques

4.1.1 Data passing between threads

When a thread A sends a continuation signal to a thread B, we say that A continues B. Continuation to another thread and recontinuation to the same thread is performed with instructions `cont` and `rcont`, respectively. These instructions are not concerned with the data passing between threads, but just send a continuation signal. The data passing between threads is done through memory with store and load instructions before sending continuations.

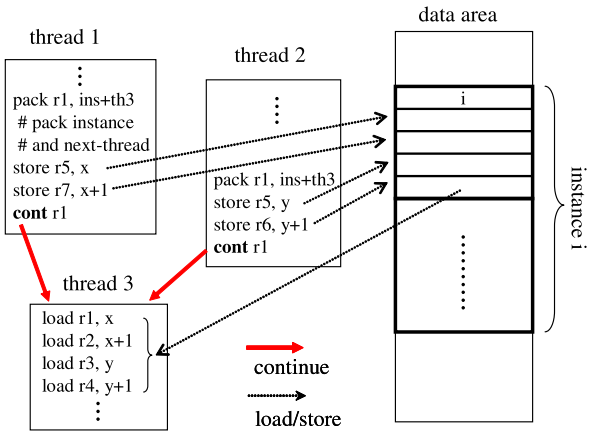


Fig. 3 Data passing between threads

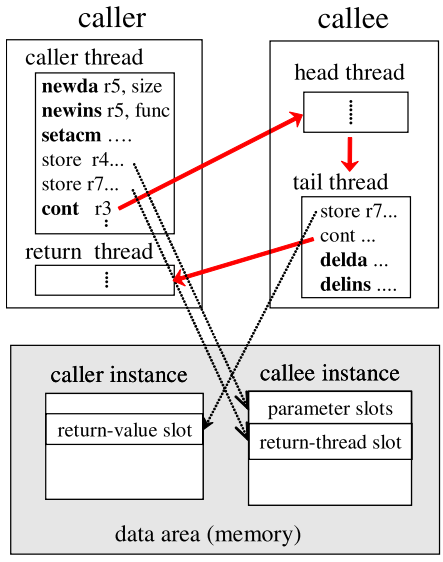


Fig. 4 Function invocation with continuations

Figure 3 illustrates the data passing between threads. Thread1 and thread2 continue thread3, after they store the values thread3 requires. When thread3 is fired, it first loads the values that thread1 and thread2 have stored.

4.1.2 Function invocation

Figure 4 shows how functions are invoked using thread continuations. Function invocation is programmed in a split-phase manner. To invoke a function, a data area of the function has to be allocated in memory. The data area consists of the parameters of the function, return value, local data and the ID of succeeding threads to be continued when the function returns. While invoking a function, the caller of the function does

not get blocked but continues its execution. A typical process of function invocation and return from the callee function is the following.

Invocation process: The caller thread of the caller function (1) acquires a new data area and a new ACM page for the callee function with macro instructions `newda` and `newins`, (2) registers all of the thread information, i.e., the set of fan-in's and code-addresses of the callee function into the newly acquired page with instruction `setacm`, (3) stores parameter values to be passed to the callee into the parameter slots in the newly acquired data area, (4) stores the ID of the thread to be continued into the return-thread slot, then (5) continues the head thread in the callee with instruction `cont`.

Return process: The tail thread of the callee function (1) stores a return value into the return-value slot in the data area of the caller function, and (2) continues the succeeding thread of the caller function with instruction `cont`, then, (3) deletes the data area and ACM page of the callee with instructions `delda` and `delins`.

4.1.3 Iteration

There are two ways to program the iteration. One way is to use conditional branch instructions within a thread. However, this method might require a long time to complete the execution of a single thread if the loop repeats many times in it. This is undesirable for effective parallel execution of multiple threads in the *Fuce* processor. Another way is to define a tail-recursive function for expressing iteration as shown in Fig. 5.

The advantage is that whenever such a function is invoked, it is not necessary to acquire a new ACM page and data area for the function, since the same ACM page and data area can be reused, and eventually, the tail-recursive function is executed as thread level iteration by the *Fuce* processor. Therefore, the execution of a tail-recursive function is quite efficient in comparison with that of normal function. In addition, by using this method, threads are frequently switched, and never occupy the processor resources for a long time.

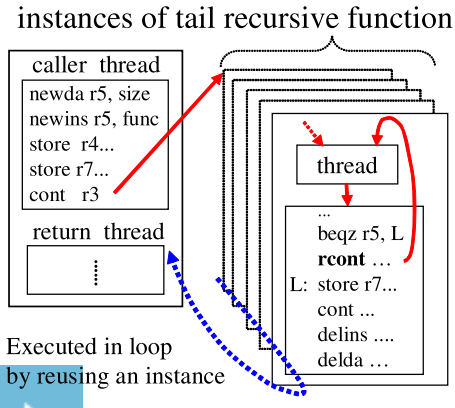


Fig. 5 Iteration by tail-recursion

4.1.4 Mutual exclusion

Mutual exclusion is necessary for non-reentrant routines such as I/O-event handlers and dynamic memory allocators in operating systems. In the *Fuce* processor, special instructions are implemented to support mutual exclusion.

- **[trylk rd, rs]** sets one to `rd` and sets the lock-bit of the target thread shown by `rs` to one if its value is zero. Otherwise, sets zero to `rd`.
- **[unlk rs]** sets the lock-bit of the target thread held in `rs` to zero.

These instructions do not access to the main memory, but just modify the lock-bit fields in ACM. This differs from conventional test-and-set operations such as LL (load linked) and SC (store conditional) in MIPS architecture [17]. Conventional test-and-set operations cause accesses to main memory, leading to a lot of pipeline stalls. On the other hand, accesses to ACM will not cause so many pipeline stalls due to a short access latency. Mutual exclusions implemented with this approach never use main memory, and thus would be very efficient.

Figure 6 illustrates threads that are implemented with the instructions described above. Suppose the fan-in value of thread0 is two. Thread1, thread2, and thread3 are trying to store a value into a memory location *x*. Only one of them will be able to continue thread0 after the thread0 has continued itself by executing the instructions `rcont` and `unlk`. Here, the instruction `trylk` is used to guard against a race condition among the requesting threads.

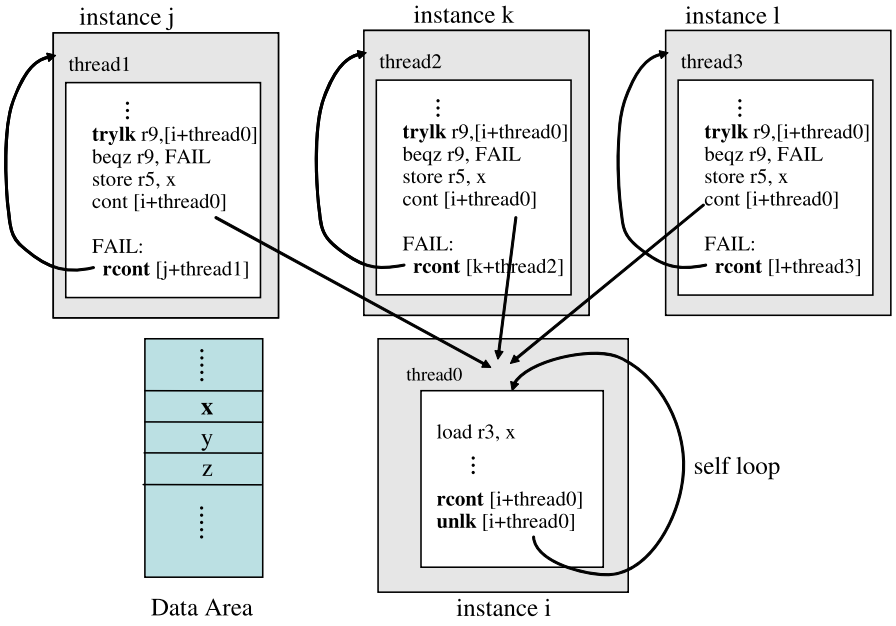


Fig. 6 Mutual exclusion

4.2 Programming techniques for parallelism

We introduce three typical but quite important programming techniques for *Fuce* which control temporal and spatial parallelisms, respectively. We show the simplicity of writing parallel programs using *Fuce*.

4.2.1 Data-parallel computation

Data parallel computation is programmed as parallel function invocation or explicit parallel thread execution. In the parallel function invocation, activated function instances compute each data element in parallel. This is called instance-level parallelism. Another approach is to execute multiple threads in parallel for each data element within an instance. This is called thread-level parallelism. Figure 7 depicts the instance-level parallel computation of a fork-join program.

4.2.2 Thread pipelining

The continuation model of *Fuce* allows us to program the thread pipelining easily. The thread pipelining provides temporal parallelism and it can be used typically for stream processing described in Sect. 6.1.

To program the thread pipelining, we make use of the instructions `cont` and `rcont`, or test&lock instructions for mutual exclusion. Figure 8 shows a version

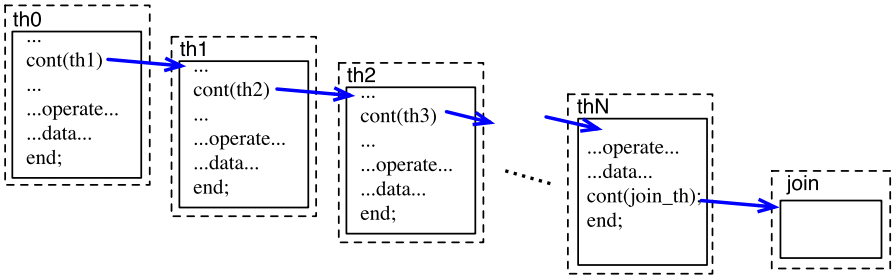


Fig. 7 Data parallel computation

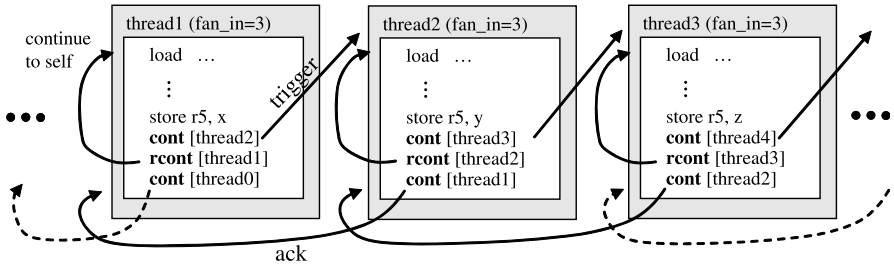


Fig. 8 Thread pipelining

of using only `cont` and `rcont`. The fan-in value of each thread is three. After continuing to itself, each thread is waiting for two continuation signals (*trigger* and *ack*) from the neighbouring threads. The instruction `rcont` is used because the threads for pipelining iterate at thread-level and have to be refired as many times as inputs exist.

4.2.3 Parallelism control

How to manage parallelism is an important problem in parallel processing. In particular, dataflow-based computation exploits parallelism beyond the limit of hardware resource. With the *Fuce* processor, parallelism can be constrained within the capacity of hardware resource, e.g., the number of waiting threads in the ready-queue or the number of active instances, by using the hardware register *hd-load* that displays the hardware load.

As functions are invoked in split-phase manner, we can invoke multiple functions at the same time. This parallel invocation provides spatial parallelism. However, when we deal with problems which have potentially high parallelism, there will be an explosion of function instances, and finally we will fail to solve the problems because of a depletion of hardware resource such as ACM.

With a little modification of function invocation, we can control the parallelism of program. The number of functions to be invoked is easily controlled by using a condition variable, as shown in Fig. 9. In the figure, *hd-load* displays the load of the hardware resource and *p* is set to some threshold value of the hardware load. Then the *Fuce* processor switches between parallel and serial invocations according to the value of *hd-load* during the execution.

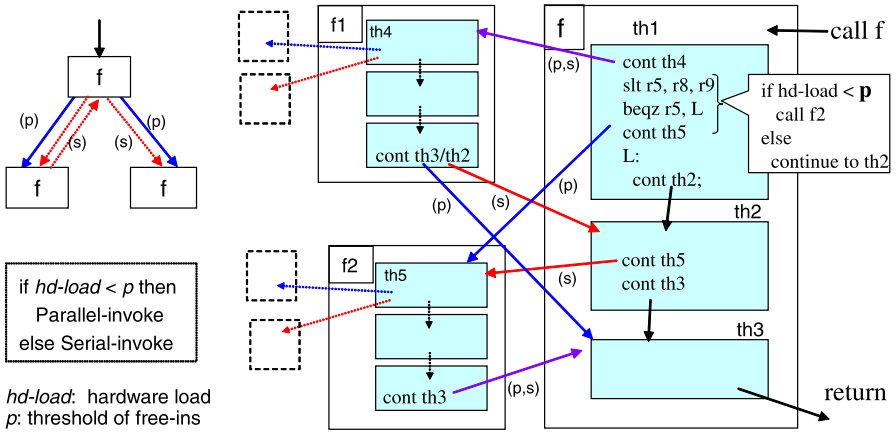


Fig. 9 Parallelism control

5 *Fuce* language system

5.1 Continuation-based multithreading language *CML*

Continuation-based Multithreading Language *(CML)* is designed for writing all kinds of programs, even including OS kernel programs, based on the concept of continuation. *CML* is basically a C language extended with several thread-related operations described below.

Function and process definition: A `function` or `process` descriptor is added to each normal function definition of C. A process differs from a function in that it continues to itself tail-recursively without returning a value.

Data-Area (DA) variables: `darea` decorator is added to a local variable if it is to be allocated to the main memory. Otherwise the local variable is allocated to a register. Formal parameters of function are implicitly declared as DA variables.

Thread definition: The thread definition begins with `thread` decorator followed by the normal function definition of C. Note that the body of a function definition contains several explicit thread definitions plus an implicit thread definition called *entry thread* whose body is written as the usual body for the normal function of C.

Fan-in specifier for a thread: A thread definition can optionally have fan-in number specifier like `<2>` just after the thread name. If the specifier is omitted, it means that the fan-in number is 1 for the thread.

Thread transfer: The statement `cont` *thread-name*`;` (or `=>` *thread-name*`;`) is used to issue a continuation to the specified thread.

Function call: A function call in *Fuce* is done in a split-phase way. That is, the called function is executed on a thread different from that for the caller. Thus, the return value has to be assigned to a DA variable instead of a register. Also, at the same time, the caller has to inform the called function of the name of the thread that will receive the return value. For this, the following syntax is used.

DA variable = function-name(parameters) => thread-name;

Self-recursive call: It should be efficient for a recursive function to reuse the same function instance (local variables and the ACM page) when recursively calling itself, even if the recursively called function has to run on a logically different thread. For this, `recur(parameters)` is used. Actual parameters will be overwritten on the formal parameter variables for the function. Also, `recur[N]` is used when a process reinvokes itself with issuing N continuations (signals).

The final statement: The final statement of a thread body should be one of `return`, a thread transfer statement, or a self-recursive call to the function.

A syntactic sugar for a channel: The following statements of Hoare's CSP like syntax are available for a channel variable. `ch ?? v` to get a value from channel `ch` and store it to `v`. `ch !! v` to put a value of `v` to channel `ch`. Also, as in `proc(c::ch<+>)`, a channel variable can be accompanied with an annotation `<+>` or `<->` to specify the channel is connected as an output or an input, respectively. In this case, process `proc` becomes a sender to channel `ch`, and `ch` is passed to formal parameter `c`.

Figure 10 illustrates a *CML* program that computes a factorial number.

5.2 Converting C programs to *CML* programs

Conventional C programs are converted to *CML* programs through the following processes. We omit here the details of the converting process (see [18]) but show the brief sketch.

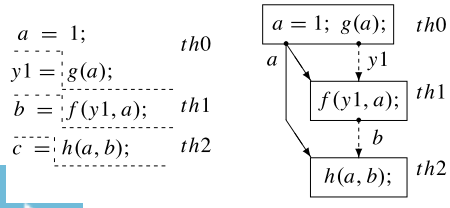
1. **Automatic thread extraction:** For any block of statement in a C program, the thread number is assigned to each component of a block, by using three variables: the base thread number, the current thread number and the next new thread number. The converting process is performed in the order of the block in assignment statement with/without function call, conditional statement, and iteration statement.
2. **Dependency analysis between threads:** Then the dependency between extracted threads is analyzed. Figure 11(right) shows the dependency graph of threads extracted from an example code fragment of C shown in Fig. 11(left). Here, solid arrows represent references to DA variables, and dotted arrows represent function value definitions.
3. **Thread fusion:** For any part of the dependency graph, threads are fused into a single thread if there is no parent and child relation between the threads, but there exits chains of variable reference relation between them.
4. **Deciding fan-in number of a thread:** The fan-in number of a thread in a program is decided according to the following theorem for the dependency graph for threads constructed from the program. (Its proof is omitted.)

Theorem 1 *The fan-in number of a thread is equal to the sum of the number of incoming dotted arrows.*

Fig. 10 *CML* code for a factorial function

```
function int fact(int n) {
  darea int m;
  thread out_fact {
    return n * m;
  }
  if (n > 0) {
    int p = n-1;
    m = fact(p) => out_fact;
  } else {
    return 1;
  }
}
```

Fig. 11 Thread extraction and a dependency graph



5.3 *CML* compiler

The *Fuce* instruction set except the thread-related instructions is based on the MIPS system and the most part of the *CML* grammar borrows the grammar of C. Therefore, it takes no much effort to implement the *CML* compiler, if *CML* programs can be translated into complete C programs by modifying the code generation part of existing C compiler to fit in the *Fuce* execution model. A *CML* program is compiled *Fuce* machine code through the intermediate code called *HAL* (*High-level Assembly Language*). *HAL* has the same syntax as the C language.

Translation of *CML* into *HAL*

Thread-related instructions shown in Table 1 are translated by using inline-assembling functions and macros. The outline of translation is as follows. DA variables declared in a *CML* function are defined as a global structure in C, and two fields are added to the DA structure for the thread ID (ACM entry number) of the caller that calls a function and the pointer to the DA variable that stores the return value. Then the thread is redefined as the function of C. At that time, the thread number `id` representing the self- thread and the pointer `da` to the structure that holds DA variables are set up as the first and second arguments of the C function.

Figure 12 shows the *HAL* code obtained by applying the above translation to the *CML* program of Fig. 10. Since *HAL* codes contain no function call, it is easy to optimize the codes.

Modifying existing C compiler

In order to generate the *Fuce* machine code, we modify the code generation part of the existing C compiler (gcc). As the *Fuce* execution model has no stack and even no notion of register save/restore, we modify the portion where stack operations, i.e., register save/restore, are performed in the code generation part of the existing C compiler (gcc). In addition, we modify the register convention of compiler so as to adapt it to *Fuce*. We assign the first argument (`id`) of C function to register #1, and

```
typedef struct _fact_darea {
    int n;
    int m;
    int return_thid;
    int *return_val;
} *fact_darea

out_fact(int id,
         fact_darea da) {
    *(da->ret_val) =
        da->n * da->m;
    cont(da->ret_thid);
    delins(id);
    delda(da);
    end;
}
```

```
fact(int id, fact_darea da){
    if (da->n > 0) {
      fact_darea *f1_da = newda(fact);
      int f1 = newins(fact, f1_da);
      f1_da->n = da->n - 1;
      f1_da->ret_thid = 1;
      f1_da->ret_val = &(da->m);
      cont(f1);
    } else {
      int ret_id = da->ret_thid;
      *(da->ret_val) = 1;
      cont(ret_id);
      delins(id);
      delda(da);
    }
    end;
}
```

Fig. 12 *HAL* code

```
thread th(ST *p) {
  int i;
  ...
  i = p->x;
  ...
}
```

(a) Before PLO.

```
thread th(ST *p) {
  int i;
  int r = p->x;
  ...
  i = r;
  ...
}
```

(b) After PLO.

Fig. 13 Preloading optimization

the second argument (`da`) to register #3. This is because, in *Fuce*, a thread execution begins with setting the thread ID in the register #1 and the address of data area in the register #3.

Preloading optimization

The use of PLU is one of the most significant advantages of the *Fuce* processor. The preloading operation for variables of a thread is done by one of PLUs while its corresponding MU is executing the body of another thread, before the execution of the thread body starts in the MU.

The preloading mechanism is very useful to hide memory access latency, and it achieves two times speedup, in the ideal case, compared to those cases without preloading.

In order to utilize the mechanism, we apply Preloading Optimization (PLO) to the given thread code. The PLO code consists of fore-part and rest-part. The fore-part includes only load instructions, whereas the rest-part has any instructions. We try to move as many load instructions as possible to the fore-part. Figure 13 demonstrates a typical example of code moves. For a code fragment `p->x` in the thread body, a new variable `r` is introduced; the code `r=p->x` is added in the fore-part; and the original code fragment is replaced with `r`. Note that `r` is mapped to a register but not the main memory.

6 Examples of *CML* code

6.1 Stream programming

Among various styles of parallel/concurrent programs, *stream processing programs* are particularly useful and suited for execution on *Fuce*. Stream processing is simply written by using continuation.

Continuation method of stream programming

Consider a pair of processes $\langle w, r \rangle$ connected by a channel ch. The process w generates a series of values for x and sends them to the process r. This stream processing can be written in *CML* as in Fig. 14. The action of w and r is controlled by cont instructions. Here we call init-, trigger-, or ack-cont for each cont instruction according to its usage.

```
typedef struct {
  int flag,value,to,from;
  proc to,from;} chan;

process main() {
  darea chan ch;
  proc w = new W();
  proc r = new R();
  ch.from = w; ch.to = r;
  w(ch<+>);
  r(ch<->);
  cont w;
}
```

```
process W(chan *ch) <2> {
  /* computing x's value */
  ch->value = x;
  cont ch->to;
  recur;
}

process R(chan *ch) <2> {
  x = ch->value;
  cont ch->from;
  /* using x's value */
  recur;
}
```

Fig. 14 Continuation method

```
process main() {
  darea chan ch1;
  proc n = new ints();
  proc s = new sieve();
  n(2,ch1<+>);
  s(ch1<->);
}

process sieve(chan *ch)<2>{
  darea chan ch2;
  int a;
  if (ch ?= nil) exit;
  ch ??? a;
  if (a>10)
    {prints(ch); exit;}
  printf(a);
  proc f = new filter();
  proc s = new sieve();
  f(a,ch<*>,ch2<+>);
  s(ch2<->);}
```

```
process ints(int i,
             chan *ch)<2>{
  if (i > 100)
    { ch !! nil; exit; }
  ch !! i;
  recur(i::i+1);
}

process filter(int e,chan *cha,
               chan *chb) <3> {
  int a;
  if (cha ?= nil) {
    chb !! nil;
    exit;}
  cha ?? a;
  if (a % e == 0) {
    recur[2]; /* issues twice - - (1) */
  } else {
    chb !! a;
    recur;}}
```

Fig. 15 *CML* code for prime number generation

When w is invoked by an init-cont or an ack-cont issued by r, w puts a value of x to ch, then issues a trigger-cont to r so that r can get the value. When r is triggered by w, r gets the value of x from ch, then issues an ack-cont to w so that w can put the next value. Note that the fan-in number for $w(r)$ is two since it requires a recur-cont plus a trigger-(ack-)cont.

Example: prime number generator

Figure 15 shows the *CML* code for generating prime numbers. `prints(ch)` will display every number that is put in `ch`. An annotation `<*>` for a channel is used to change the output channel to another for putting values. Thus, when the input channel `ch` for `sieve` is passed onto an instance `f` of the filter process `filter`, its destination (`ch.to`) is changed from `sieve` to `f`. This makes it possible to send data directly from `ints` process to `filter` process. Similar change will be done when

Fig. 16 *CML* code for merge

```
process main() {
  darea chan ch1,ch2,ch3;
  int p = new producer1();
  int q = new producer2();
  int r = new merge();
  int s = new consumer();
  p(0,ch1<+>);
  q(1,ch2<+>);
  r(ch1<->,ch2<->, ch3<+>);
  s(ch3<->);
  cont p; cont q;
}

process merge(chan *cha,chan
          *chb,chan *chc) <3> {
  darea int i=0;
  int u,v;
  if (i==0) u = cha->upd;
  else u = chb->upd;
  if (i != u) {
    cha->upd = 0;
    cha ?? v; chc !! v;
  } else {
    chb->upd = 0;
    chb ?? v; chc !! v;
  }
  recur(i::(i+1)%2);}
```

Fig. 17 Resource manager

```
process resman(chan *req,
               proc man) <2>{
  proc procid;
  req ?? procid;
  if (procid!=NULL) {
     /* enqueue procid */;
  }
  unlock(man);
  recur;
}

process resuser(chan *req,
                proc man) {
  ...
  proc p = new resuser_body()
  if (!lock(man)) recur;
  req !! p;
}
```

`filter` process on the upper stream wants to connect itself to another `filter` on the lower stream.

Note that two recur-conts are issued at (1) in the figure. When `a%e` is equal to zero, no `ack` is issued by `chb.to`, since no data will be put to `chb`. To fake as if a data is sent to `chb.to`, an ack-cont is issued.

Other examples

Other *CML* sample programs are shown in Figs. 16, 17, and 18.

Figure 16 shows a merge process. The merge process `r` gets the stream of even-numbers (generated by producer `p`) or odd-numbers (generated by producer `q`) via channel `chp` or channel `chq` selectively, and puts the merged stream to consumer `s` via channel `chc`. (Codes of producer and consumer definitions are omitted in Fig. 16.)

Figure 17 shows a shared resource management program used as a monitor for arbitrary number of processes. Lock/unlock operators are available in *CML* to realize mutual exclusion. Note that the lock/unlock operations is basically equivalent to the traditional test&set instruction, except for the lock/unlock operation is performed to an ACM entry, that is, a thread. To each monitor an input channel is associated. When any process requests to the monitor for accessing the shared resource, it issues a lock instruction to the monitor process, and other competing processes requesting the same monitor will busy-wait until its lock instruction completes.

6.2 Pipelining iterative codes

The stream programming in *CML* can be applied to pipelining a nested loops such as the following:

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    V[i] += A[i][j] * U[j];
```

Its pipelining *CML* code is shown in Fig. 18. When the process `pipeline` is invoked at the first time, it creates and invokes a new process `task` for $j = 0$; sets up channel `out` linking the two processes; and `recurs` to itself. In each recursion of `pipeline`, it puts the current value of `i` to `out` and increments `i` by one.

`Task` process for each value of j $(0 \leq j \leq N-1)$ performs the following.

1. It gets `i` and `sigma` from the channel `in`.
2. When it is invoked at the first time, it creates and invokes another `task` for `j+1`, then sets up channel `out` linking the two `task` processes.
3. In each recursion of `task`, if $j \leq N$,
   it puts the value `sigma+A[i][j]*U[j]` to `out`, and issues a trigger-cont to the succeeding `task` whose ID is `out.to` and issues an ack-cont to the preceding `task` whose ID is obtained from `out->from`, then recurs to itself.

```
struct chan {int flag,
                 val, val1;
             proc to, from;}

process pipeline(int i, int j,
                 int V[],int U[],
                 int A[][]) <2> {
  darea chan out;
  if (!out.to) {
    proc p = new task();
    out.to   = p;
    out.from = id;
    p(out, j, V, U, A);
  }
  if (i < M) {
    out.val = i;
    cont out.to;
    ++i;
    recur;
  }
}
```

```
process task(chan *in, int j,
             int V[],int U[],
             int A[][]) <3> {
  darea chan out;
  int i     = in->val;
  int sigma = in->val1;
  if (!out.to) {
    proc p = new task();
    out.to=p; out.from=id;
    p(out, j+1, V, U, A);
  }
  if (j < N) {
    out.val  = i;
    out.val1 = sigma +
      A[i][j] * U[j];
    cont out.to;
    cont in->from;
    recur;
  }
  V[i] = sigma;
}
```

Fig. 18 A pipeline code in *CML*

Otherwise, it writes the value `sigma` to `V[i]`, then terminates. Here, only the final `task` ($j = N$) performs this action.

A sequential computation for the original code would take $O(M * N)$ time. On the other hand, provided an ideal parallel execution environment, the pipelined code would take just $O(M + N)$ time, that is $O(N)$ time for completing the pipeline construction and $O(M)$ time for completing summation at `task` for $j = N$. Thus, significant speedup can be achieved.

7 Evaluation of *Fuce* architecture

Performance of the Fuce processor is evaluated using the *Fuce* software simulator. The software simulator executes the multithreaded programs written in *CML* and simulates the behavior of Fuce hardware modules shown in Fig. 2 in clock cycle level.

7.1 Execution time versus memory access latency

First, we measured the execution time of benchmark programs with the following simulation parameters: Number of TEUs = 8, Memory Access Latency = 10 ∼ 120 clock cycles, Preload: off, BR: off. Figure 19 shows the measured execution time in clock cycles. As the figure shows, the execution clock cycles increase about 2-fold∼4-fold against a 12-fold increase in the memory access latency.

7.2 Effect of preloading

We measured the effect of preloading. Figure 20 shows the speedup achieved with the use of PLU. The figure shows that the speedup of 1.2 to 2.0 times is obtained by using PLU. Furthermore, the execution time of thread codes optimized for preloading is compared to the execution time of the unoptimized program code that executes load instructions on demand. Figure 21 shows the optimization is effective.

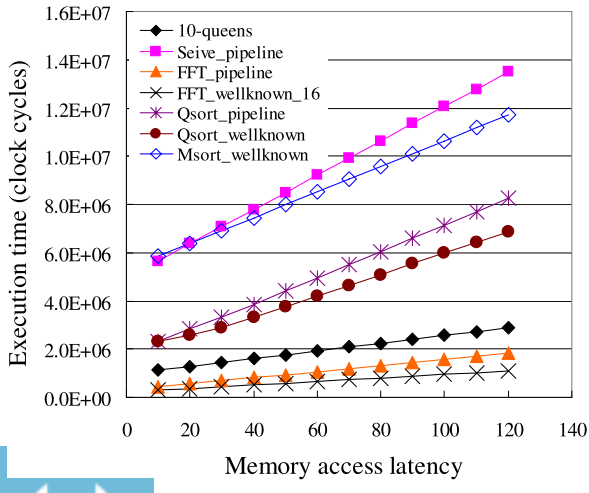


Fig. 19 Execution time of benchmark programs

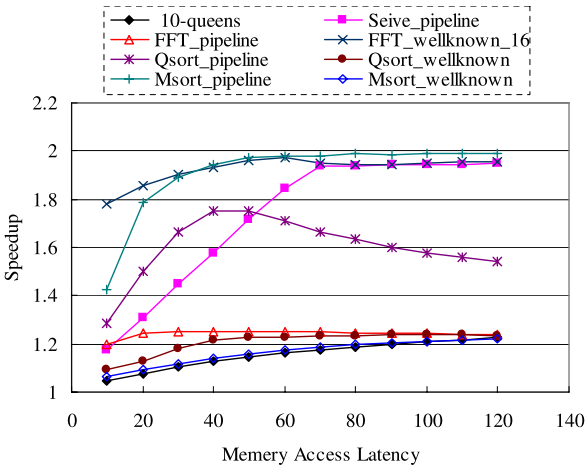


Fig. 20 Effect of preloading unit

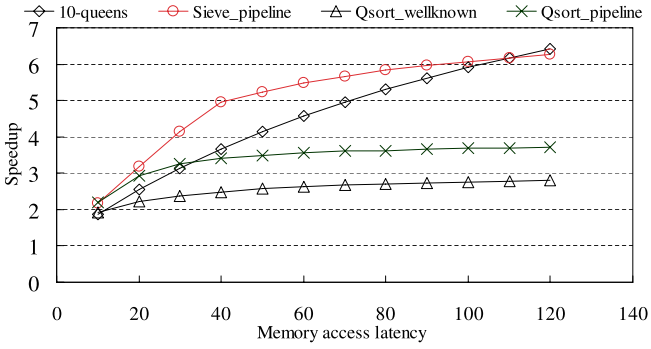


Fig. 21 Comparison of execution time between preloading thread codes and conventional codes

7.3 Effect of bridge registers

We measured the effect of BR in the execution time. Figure 22 plots the speedup achieved when BR are used in the inter-thread data passing. The figure shows that the effect of BR is more explicit for larger memory latency. Figure 23 shows that the execution time is less sensitive to BR access latency.

7.4 Performance sensitivity of ACM access latency

We measured the affect of ACM access latency on the execution time. Figure 24 depicts the relative performance on different ACM access latencies. The figure shows that the execution time is less sensitive to ACM access latency.

7.5 Effect of parallelism control

We measured the effect of parallelism control described in Sect. 4.2.3. Figures 25 and 26 show the performance change when the value of *hd-load* is set to the number of

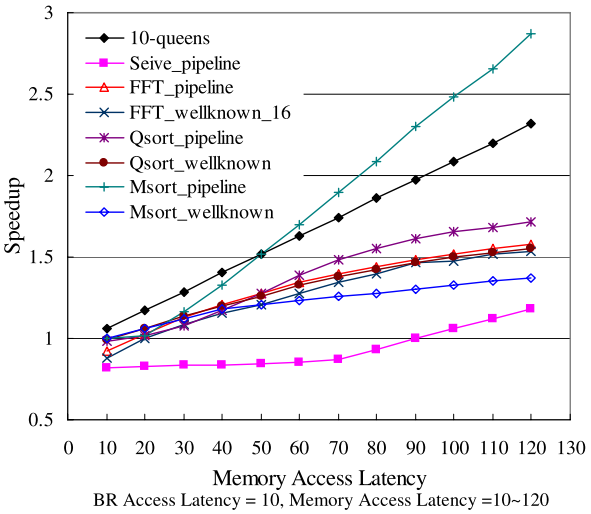


Fig. 22 Speedup by bridge registers

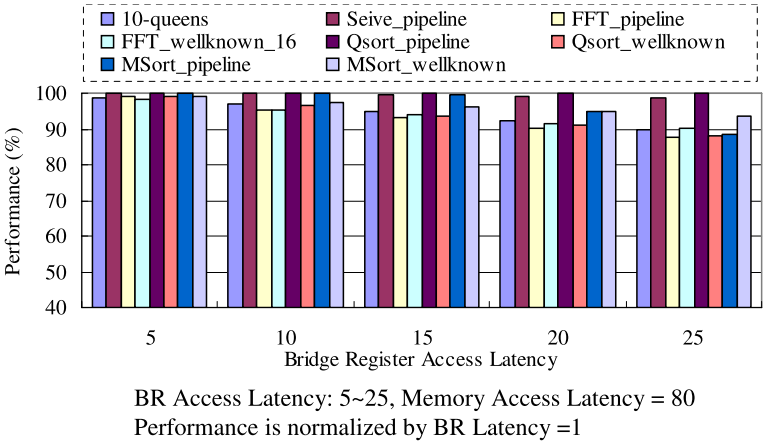


Fig. 23 Relative performance on different BR access latencies

waiting threads in the ready-queue. A similar pattern was observed when the value of *hd-load* was set to the number of active instances. Figure 25 shows that more than 90% of performance is achieved even if the number of waiting threads in the ready-queue is set to a small value, 16, for the 10-queen program. As Fig. 26 shows, the performance hardly changes against the threshold value of 4-64 for the quick sort program.

7.6 Speedup against the number of TEUs

We measured the potential of the *Fuce* processor to exploit the parallelism of multi-threaded programs. Figure 27 plots the speedup against the increase in the number of

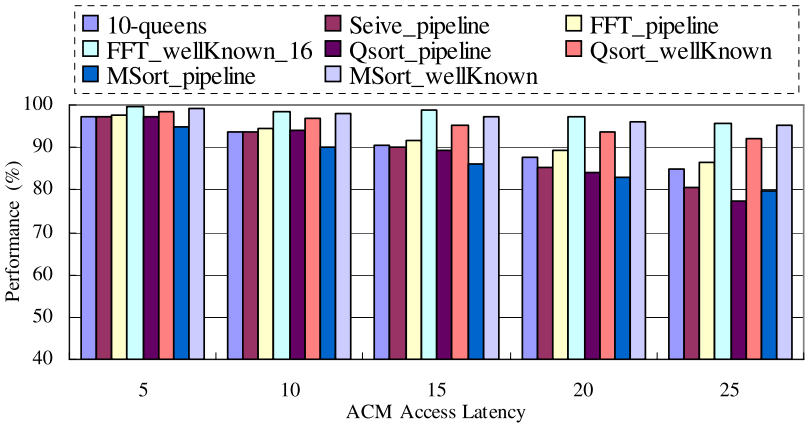


Fig. 24 Relative performance on different ACM access latencies

Fig. 25 Effect of parallelism control for 10 queens

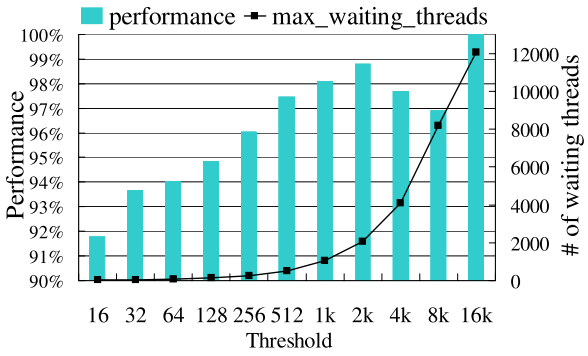
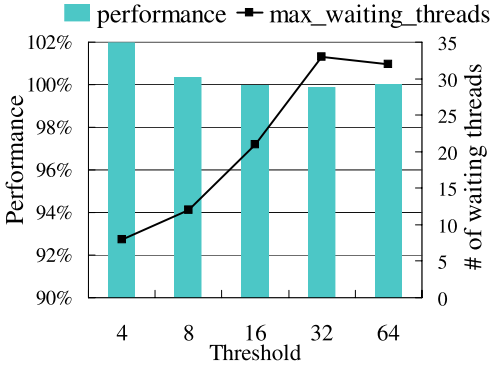


Fig. 26 Effect of parallelism control for quicksort



TEUs. The figure shows that the *Fuce* processor has enough potential to exploit the parallelism of given multithreaded programs.

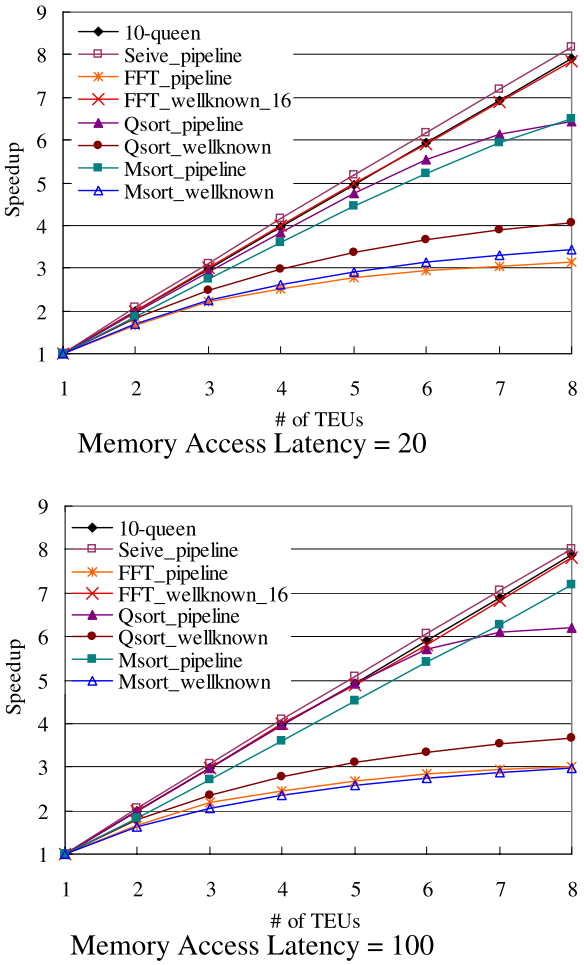


Fig. 27 Speedup against the number of TEUs

7.7 Effect of pipelined iterations

We evaluate the effect of pipelining iterative codes described in Sect. 6.2. Figure 28 shows the speedup between the time taken for the serial code (the nested loop shown in Sect. 6.2) and the time for the pipelined code (shown in Fig. 18) against the number of TEUs. Here, we used two kinds of pipelined programs: `pipeln` is exactly the same version as in Fig. 18 and `pipeln2` is a larger-grained thread version, in which each `task` process contains two multiply expressions for calculating two array elements. Both are optimized for the preloading. Experiments were made with simulation parameters: preload=ON, no. of TEUs=1-8.

We can see that the pipelined programs in *CML* exploit the pipeline parallelism considerably. Especially, the program `pipeln2` achieves mostly ideal speedup (of over 7x with 8 TEUs). The reason is that the preloading works more effectively on the execution of larger-grained threads than on that of smaller ones.

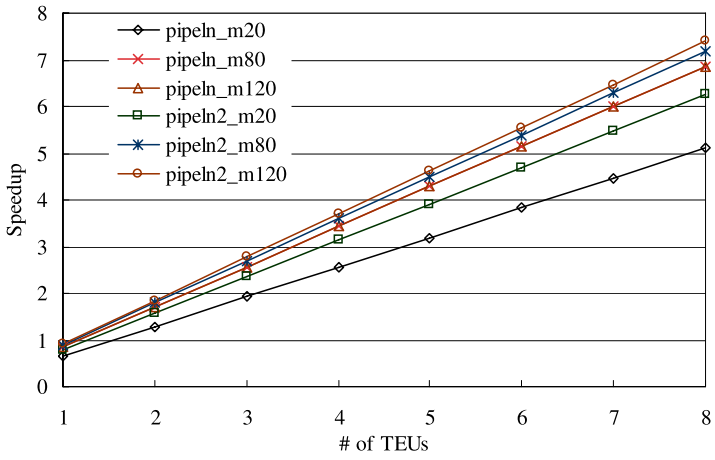


Fig. 28 Performance of pipelining iterations

These benchmarks, though small, give evidence that pipeline parallelism is extracted by stream programming and the *Fuce* execution mechanism can exploit pipeline parallelism from sequential nested-loop codes.

8 Conclusion

This paper has described a language *CML*, its compiler, and a concurrent programming technique for an innovative architecture *Fuce* which is based on noninterruptible threads. Perpetual processes are realized by recursive threads, process communication by a data structure called channel. The lock/unlock instructions in *Fuce* is as powerful as test&set and used for realizing mutual exclusion of shared resources. A cont instruction, lock/unlock instructions, and channels make it possible to write various ways of stream processing programs. Also, a resource management program necessary in an OS implementation can be written in the similar manner by implementing a waiting-queue for multiple processes.

From the viewpoint of description level, *CML* programs seem rather in lower level. However, *CML* has an enough capability for writing data synchronization and offers a high level of abstraction functions for writing stream processing programs. Moreover, mutual exclusion and synchronization can be written much more easily compared to other multithread oriented languages such as Java. From the viewpoint of programming paradigm, the simple style of programs based on noninterruptible threads and continuation is quite innovative. Though a good programming discipline would require some time of practicing with the new language *CML*, the gain obtained would be enormous. Finally, it is quite challenging to apply this paradigm to implementing all levels of programs ranging from an OS to applications.

References

1. Kongetira P, Aingaran K, Olukotun K (2005) Niagara: A 32-way multithreaded sparc processor. IEEE Micro 25:21–29
2. Gochman S, Mendelson A, Naveh A, Rotem E (2006) Introduction to Intel core duo processor architecture. Intel Technol J 10:89–97
3. Amamiya S, Izumi M, Matsuzaki T, Amamiya M (2006) The fuce processor: The execution model and the programming methodology. In: The 2006 intl conf on parallel and distributed processing techniques and applications, 2006, pp 485–491
4. Amamiya S, Izumi M, Matsuzaki T, Hasegawa R, Amamiya M (2007) Fuce: The continuation-based multithreading processor. In: Proceedings of the 4th international conference on computing frontiers 2007, pp 213–224
5. Sakai S, Yamaguchi Y, Hiraki K, Kodama Y, Yuba T (1989) An architecture of a dataflow single chip processor. In: ISCA '89: proceedings of the 16th annual international symposium on computer architecture. ACM Press, New York, pp 46–53
6. Nikhil RS, Papadopoulos GM (1992) Arvind: *T: a multithreaded massively parallel architecture. SIGARCH Comput Arch News 20:156–167
7. Hum HHJ, Maquelin O, Theobald KB, Tian X, Tang X, Gao GR, Cupryk P, Elmasri N, Hendren LJ, Jimenez A, Krishnan S, Marquez A, Merali S, Nemawarkar SS, Panangaden P, Xue X, Zhu Y (1995) A design study of the earth multiprocessor. In: PACT '95: Proceedings of the IFIP WG10.3 working conference on parallel architectures and compilation techniques, Manchester, UK, IFIP Working Group on Algol, 1995, pp 59–68
8. Kawano T, Kusakabe S, Taniguchi R, Amamiya M (1995) Fine-grain multi-thread processor architecture for massively parallel processing. IEEE Press, New York, pp 308–317
9. Lo JL, Eggers SJ, Emer JS, Levy HM, Stamm RL, Tullsen DM (1997) Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. ACM Trans Comput Syst 15:322–354
10. Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, Upton M (2002) Hyper-threading technology architecture and microarchitecture: a hypertext history. Intel Technol J 6:1 (online journal)
11. Kalla R, Sinharoy B, Tendler JM (2004) Ibm power5 chip: A dual-core multithreaded processor. IEEE Micro 24:40–47
12. Sigmund U (1996) U.T.: Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip. In: Proceedings of the 4th PASA workshop on parallel systems and algorithms, 1996, pp 147–159
13. Roh L, Najjar WA (1995) Analysis of communications and overhead reduction in multithreading execution. In: Proceedings of the 1995 international conference on parallel architectures and compilation techniques, 1995
14. Kavi KM, Youn HY, Hurson AR (1997) PL/PS: A non-blocking multithreaded architecture with decoupled memory and pipelines. In: Proceedings of the fifth international conference on advanced computing (ADCOMP '97), Madras, India 1997
15. Ungerer T, Robič B, Šilc J (2003) A survey of processors with explicit multithreading. ACM Comput Surv 35:29–63
16. Sankaralingam K, Nagarajan R, Liu H, Kim C, Huh J, Burger D, Keckler SW, Moore CR (2008) Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In: Proceedings of the 30th annual international symposium on computer architecture, 2003, pp 422–433
17. MIPS Technologies, MIPS32 architecture for programmers, vol II: The MIPS32 Instruction Set
18. Amamiya S, Hasegawa R, Fujita H, Amamiya M (2007) A language design for non-interruptible multithreading environment fuce. In: ISC award winning paper, international supercomputer conference, Proceedings of ISC2007, 2007